PROGRAMMING IN BASIC

AN INTRODUCTION

By

SYED SHAHABUDDIN

MBA (KAR), MBA (USA), Ph.D. (USA), CDP, CDE, CSP
Professor of
Management Information Systems
and Management Science

Rs 80/2

PROGRAMMING IN BASIC

AN INTRODUCTION

PROGRAMMING IN BASIC

AN INTRODUCTION

By

SYED SHAHABUDDIN

MBA (KAR), MBA (USA), Ph.D. (USA), CDP, CDE, CSP
Professor of
Management Information Systems
and Management Science

Ferozsons (Pvt.) Ltd.

LAHORE—RAWALPINDI—KARACHI

Shahabuddin, Syed

Programming in Basic—
An Introduction

ISBN 969 0 10004 1

Printed at Ferozsons (Pvt.) Ltd., Lahore, Pakistan

TABLE OF CONTENTS

CHAPTER 3. COMPUTER PROGRAMMING PROCEDURES

CHAPTER 4. DATA INPUT - Input and Read/Data Statements

CHAPTER 5. STRUCTURED PROGRAMMING AND THE CONDITIONAL APPROACH

CHAPTER 9. FILE MANIPULATION

Preface

This book provides an introduction to BASIC programming language. The book is systematically organized to help in the learning of the language. This book could be used by anyone. That is, it does not require any prior knowledge of a computer language or computer hardware. It starts out with a general discussion of computer hardware and software, and then develops the language starting with the basic rules and then developing applications of the language using business and non-business problems.

In this book, every BASIC programming concept is developed and discussed using programming example, and practical applications. In addition, the concepts of structured and unstructured programming are discussed. The book first discusses the rules of BASIC language and then presents the programming techniques. After presenting the programming rules and programming techniques, the book covers INPUT, READ, PRINT, GOTO, FOR...NEXT, and logical statements. It also discusses extensively the creation and the manipulation of computer files.

Unlike other computer languages, BASIC is not standardized. That is, each computer vendor develops its own language rules. Therefore, the BASIC used on an IBM system, for example, may not follow the same rules as that of the Control Data Corporation (CDC) system. This is even true of microcomputers. Each BASIC uses different rules. As a result, no book could cover completely the rules needed for each BASIC on the market. However, this book covers the most common versions of the BASIC language and, in some cases, also provides a list of possible variations among BASIC. Hence, it will be advisable to use the features covered in this book on your system; and if they do not work, check your system's manual for an appropriate method.

Syed Shahabuddin

CHAPTER 1

COMPUTER SYSTEMS

INTRODUCTION

In the 1960's and 1970's, the world was in the midst of a computer revolution that overshadowed the Industrial Revolution. The Computer Revolution has already affected everyday life and has greater potential to impact lives of people in many different ways, predictable and unpredictable. The effects on everyday life can be seen in many ways. In education, computer-assisted instructions (CAI) are becoming common. The CAI allow a student to learn at his own pace without requiring the assistance of a teacher. Many medical diagnoses are done through computer. Many bank transactions are conducted through computer without human intervention such as bank teller machines, electronic fund transfer etc. The possibility that in the future robots will be performing most of the work done by humans has been made possible by computers. That is, the field of artificial intelligence (i.e. computers think as humans) has made this phenomenon possible.

The term <u>computer</u> is used so casually that many people do not know what it means. Businesses use computers for payroll, billing, inventory, accounting, personnel, customers' records, and information systems. Computers have become an integral part of business, commerce, industry, and education. Computers come in a variety of shapes, sizes, and prices to meet any need.

Since the late 1970's, microcomputers have impacted social and educational systems. That is, microcomputers have found widespread acceptance in many areas of educational, industrial, and home environments. Microcomputers cost as

little as $300 to as much as $6000. On the other hand, large, medium, or mini computers cost between $10,000 and $1,000,000.

COMPUTER SYSTEMS COMPONENTS

Each computer system use mentioned requires both the computer and several output and input devices as well software.

Hardware

Hardware is the name of the the physical devices that comprise the computer system: the central processing (CPU), the input devices, the output devices, and the secondary storage devices. Hardware is not operating systems or other software (programs). Hardware includes all parts of the computer one can see.

Computer hardware consists of the following four components:

1. Central Processing Unit (CPU)
2. Input Units
3. Output Units
4. Secondary Storage Devices

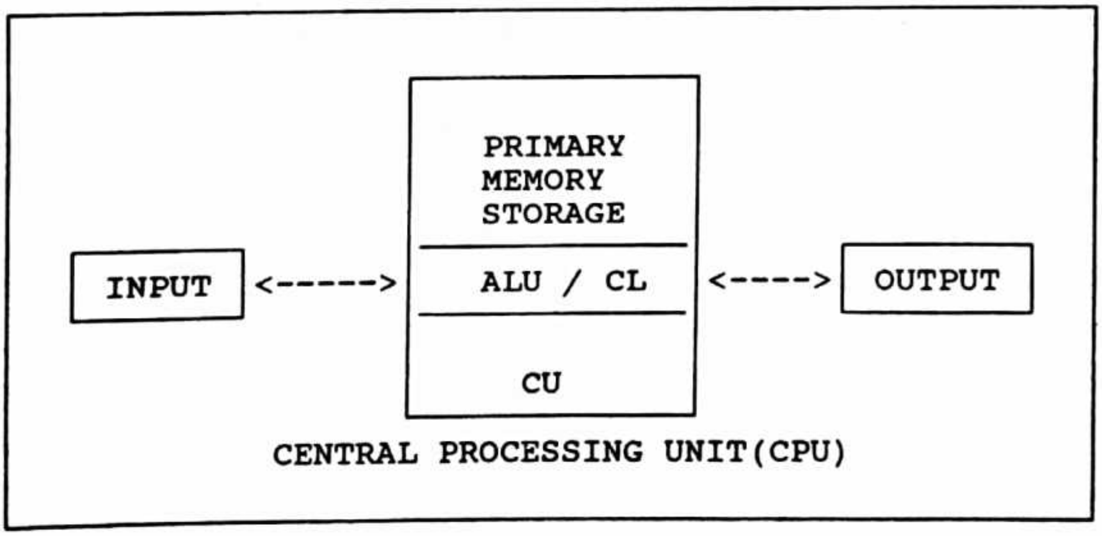


Figure 1.1 : Computer System Configuration

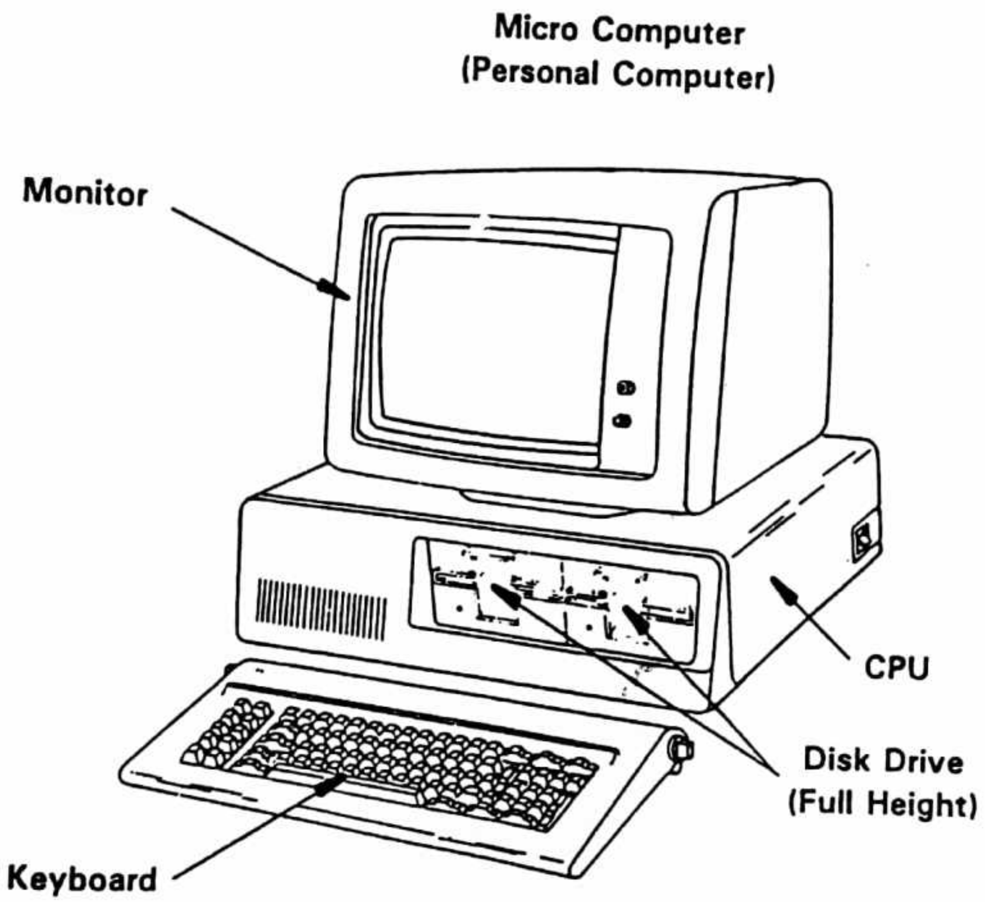


Figure 1.2 : Personal Computer (PC)

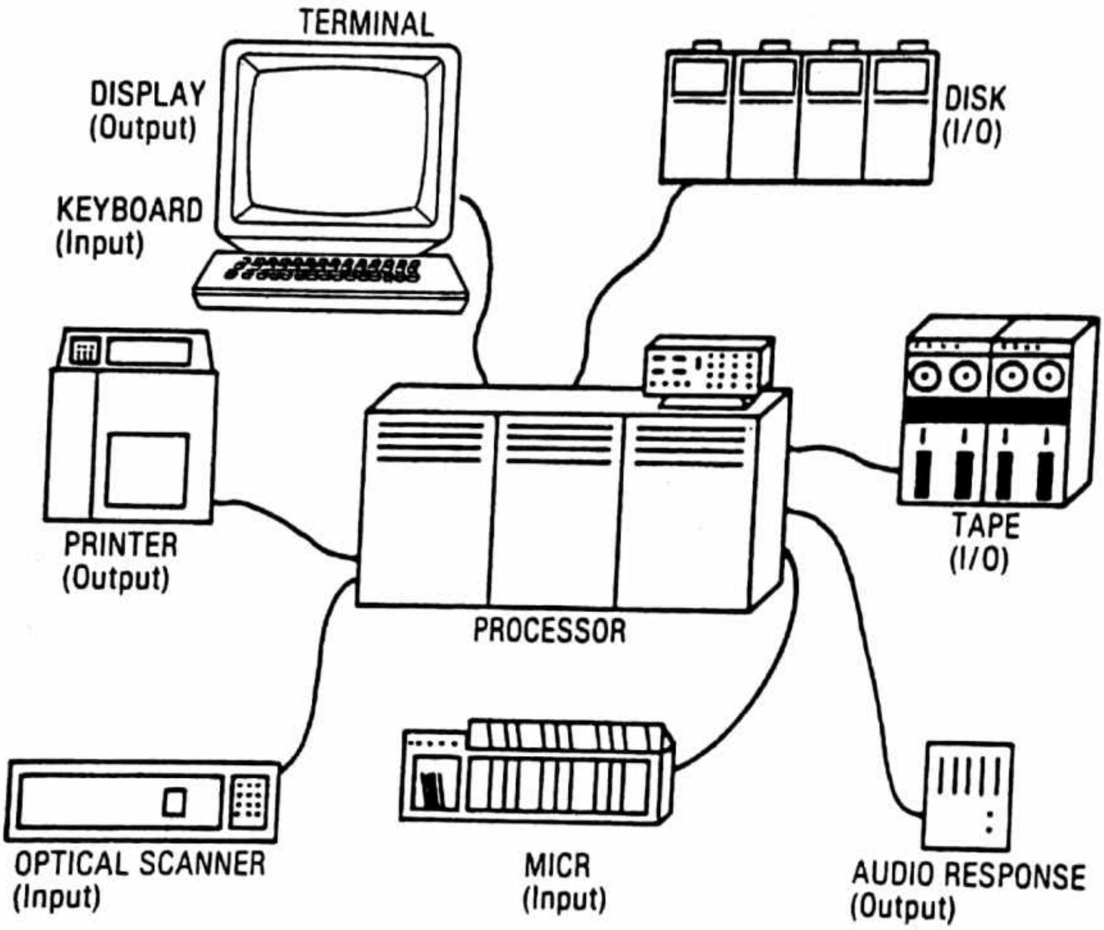


Figure 1.3 : Large Computer (mainframe)

The **central processing unit (CPU)** is like the brain of a human being. The CPU is composed of three separate units -- the control unit (CU), the arithmetic/logic unit (ALU), and the primary storage unit (memory). The control unit tells the computer what to do. That is, it controls the operations of the computer. It reads the instructions from the program and tells the other components of the computer what to do. It instructs the appropriate input device to send the needed data. It keeps a record of what parts of the program have been executed and which are still to be done. It informs the arithmetic/logic unit of calculations that need to be performed. It instructs the computer to store data in the primary memory or secondary storage. It instructs the computer to print the output on the designated output devices.

The **arithmetic/logic unit (ALU)** executes arithmetic and logic statements. A logic statement makes a comparison of two or more conditions and then performs action based on the result. Based on the instruction of the data, it performs arithmetic calculations and temporarily stores the data and the results.

The **primary storage unit** temporarily stores programs and data in the internal memory of the computer. This unit stores the program that is being executed as well as data needed by the program. It also stores intermediate results of any calculations as well as the output to be printed when the output device is not ready. This memory is different for the external memory, called **auxiliary storage**. The internal memory is part of the actual internal hardware of the computer. A computer cannot perform without a primary memory. A CPU can access only primary storage.

The size of computer memory is described in K's (Kilobytes); a K is 1,024 bytes (characters). A byte is a group of bits (<u>b</u>inary dig<u>it</u>) that forms one character. A character is a letter, digit, or special symbol. Some systems use American Standard Code for Information Interchange (ASCII) codes to form a byte, while IBM systems use extended binary coded decimal interchange code (EBCDIC) for forming a byte (see Appendix A). A computer with a CPU which could store a large quantity of information (50-million characters

or more at a time) is considered a large (or mainframe) computer. Minicomputers usually can store less than 50 million characters at a time. Most microcomputers have the capability to store somewhere between 640 thousand (640K) to four million (4 mega) characters at a time. A microcomputer with 640K has 640 thousand bytes memory. A computer of this size can store up to 640 thousand characters at a time.

There are two types of internal memories: ROM and RAM. **Read-Only memory (ROM)** is a memory chip which the programmer cannot change, because it has built-in software (called firmware) which operates the computer or performs some built-in functions. ROM instructions are hard-wired and, therefore, cannot be changed or deleted without a physical change in the circuits. For example, the IBM PC has a built-in BASIC Input-Output System. Therefore, if a PC is turned on without its operating system, the PC will get into BIOS directly. A version of ROM, called programmable read-only memory (PROM), can be programmed because it is not hard-wired. PROM is either programmed by the manufacturer or it can be shipped blank to be programmed by a user.

On the other hand, **Random-Access Memory (RAM)** is the major type of memory used in primary storage. It is used by the computer to store new software or information. It does not store information permanently. Once a new program is read into the computer, the old programs and data files are erased. Also, when the computer is turned off, all the software and data files, except those in ROM, are erased.

AUXILIARY DEVICES

Secondary Storage

Computers have to store information for future use. Microcomputers use diskettes, hard disks, cassette tape, and ROM to store information. The most common storage devices used for storing the output of a program are :

a. Disk Drives

i) Floppy diskettes
ii) Hard disks
iii) Disk packs

b. Tapes

i) Magnetic Tapes
ii) Cassette Tapes

Floppy diskettes, used only in microcomputers, can store information permanently (unless erased). The diskettes come in 3 sizes -- 3 1/2, 5 1/4 or 8 1/2 inches. Most of the PC diskettes are 5 1/4 or 3 1/4 inches. Each size can store up to 360K, 720K, 1.2MB. Each size comes in either a double-sided double density (DSDD) or a single sided double density (SSDD). Density refers to the number of bytes a disk can store on a single track. A track is a circular portion of a disk on which data can be written. The PC's DS diskette writes 40 tracks per side. Obviously, to use diskettes, the PC must have a disk drive. The disk drive comes in half-height or full-height. Regardless of the height, it can store the same amount of information on a diskette. The advantage of the half height drive is that a user could put two half-height disk drives on one side and an internal hard drive on the other side in a computer.

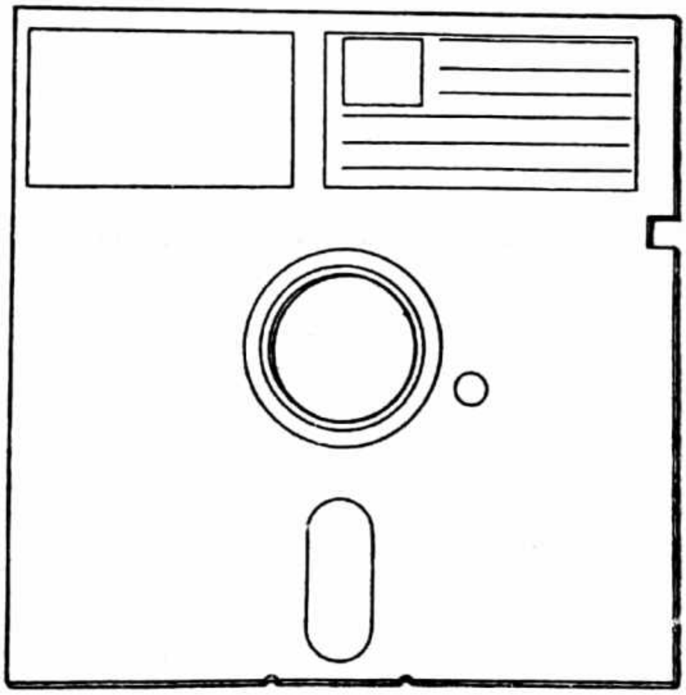


Figure 1.4 : 5 1/4 inches Diskette

Hard disks, commonly used with microcomputers, come in 20, 30, 40, or 50 million bytes (MB) and can be installed outside or inside of the computer.

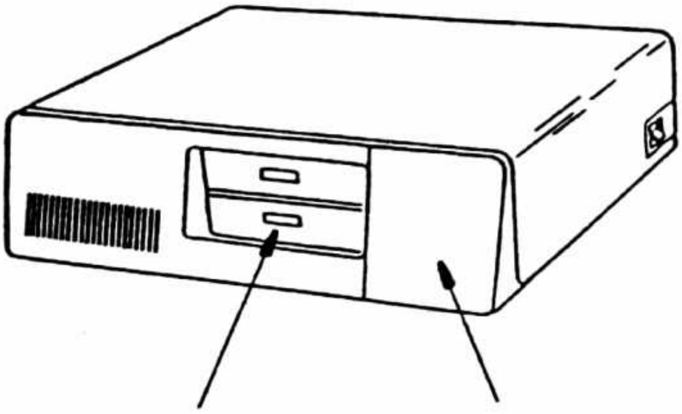


Figure 1.5: Hard Disk with Disk Drives

A magnetic disk is a metal platter coated on both sides with magnetized material. A magnetic disk resembles a phonograph record with smooth surface instead of groves. It operates in a way very similar to the use of phonograph record. That is, the disk rotates while a read/write head is positioned above the magnetic surface. The data is stored as a magnetized spot on each track. A track is concentric circle on each surface. A typical disk has 200 tracks on each surface.

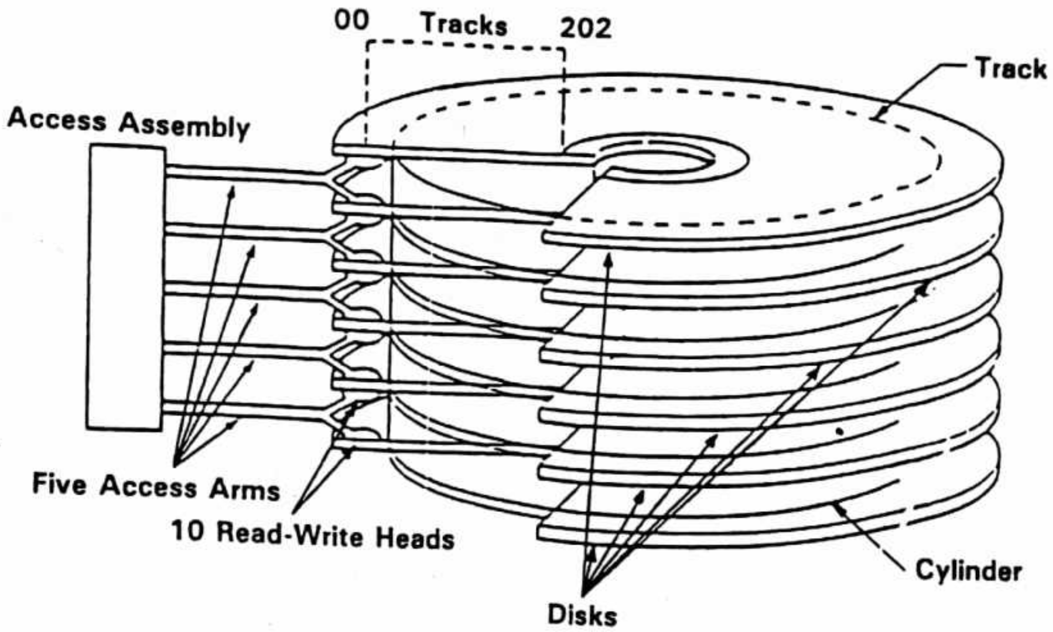


Figure 1.6 : Disk Pack

Many magnetic disks are assembled to form a **disk pack** by mounting them on a center shaft with slight space between disks to provide for the movement of the read/write head. Each disk pack usually contains six to 11 disks. Each disk surface has its own read/write head for reading or writing data on the surface. Each track is accessed by positioning the read/write head on the track. After properly positioning the read/write head, the data can be written or read from the track.

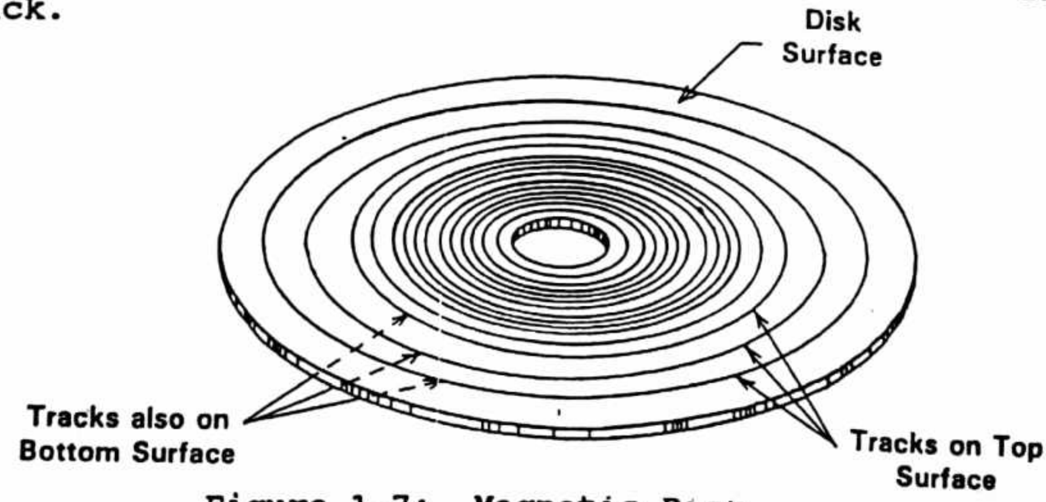


Figure 1.7: Magnetic Disk

Magnetic Tape is a continuous strip of coated plastic tape wound onto a reel. Normally, the tape is 1/2-inch wide and can be purchased in length of 600 , 1200, or 2400 feet. An inch of tape could store 1600 bytes. That is, 600 feet of tape could store up to (600 x 12 x 1600) 11,520,000 bytes per tape without any gaps between data records. Magnetic tape can be used only on large or mini computers.

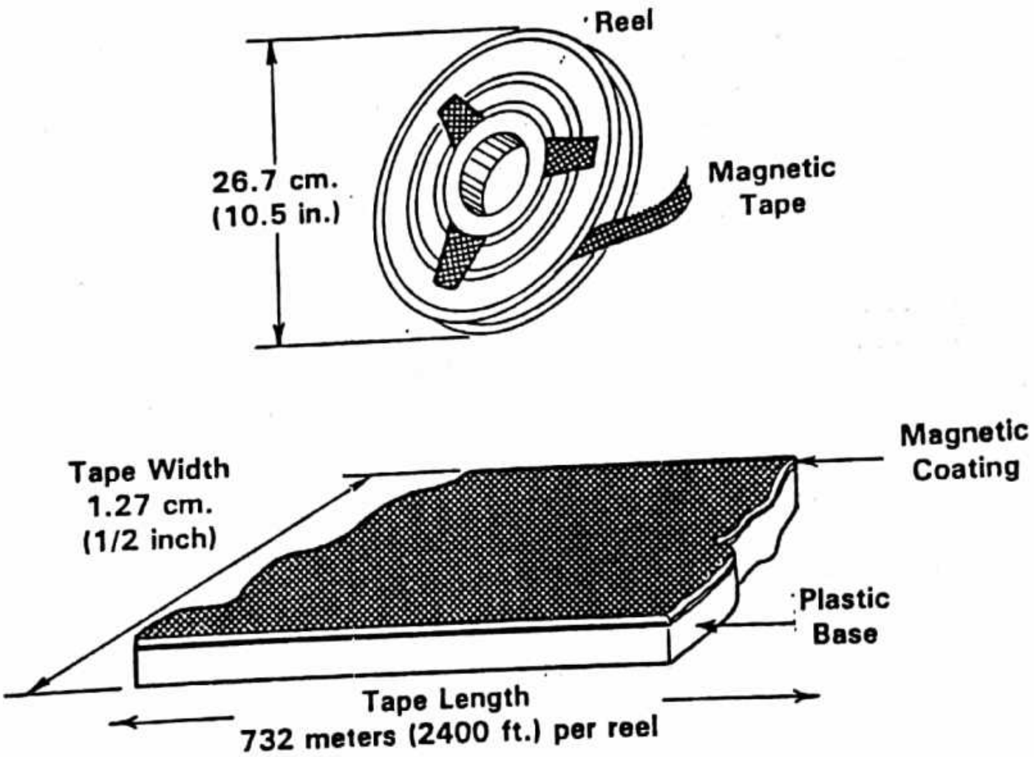


Figure 1.8 : Magnetic Tape

Cassette tapes, used commonly on microcomputers, can also store information. They look like those used in audio recording. However, this is not commonly used.

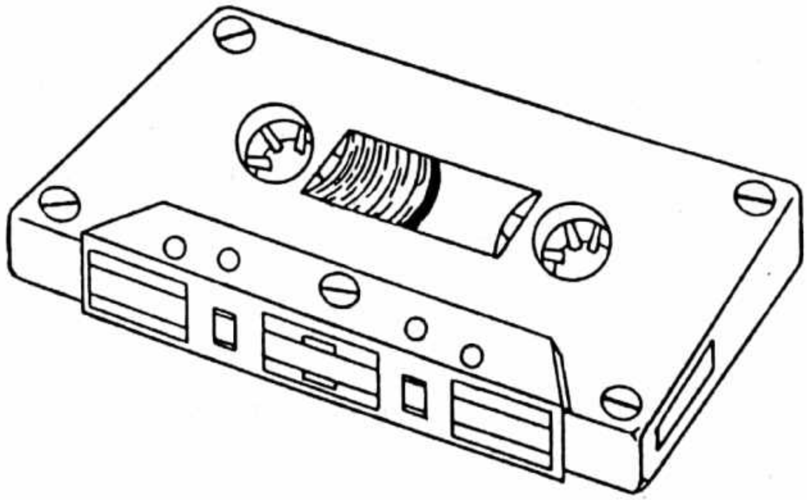


Figure 1.9 : Cassette Tape

Output Devices

The output of the processed data can be presented in several ways that suit the user's needs. That is, it can be presented on the monitor, called soft copy, or printed, called hard copy. The users should be able to determine which output device best suits their needs.

One of the output devices is the **printer**. Printers can be used to print out information and come in a variety of qualities, speeds, and sizes. Among the printers, there are dot-matrix, chain printers, and laser printers, to name a few. Each type of printer serves certain uses.

An output device, called **visual display**, is commonly used for most output. That is, information sent from the computer is displayed on monitor. A **monitor (CRT)** is an essential part of a computer. The display terminals are equipped with a keyboard which allows one to see what is inputted or what the computer outputs. The display devices can be **dumb** or **intelligent** terminals. Intelligent terminals

allow editing and formatting data entry, while dumb terminals simply accept what is inputted or outputted. Display terminals can be cathode ray tube (CRT) or non-CRT. The CRT uses television-like presentation, while non-CRT use light-emitted diode (LED) display. There are a variety of monitors available to choose from. They come in color, color/graphic, monochrome or monochrome/graphic.

Input Media

Computers can accept input from the following media:

1. Keyboards
2. Disks
3. Cassette Tapes
4. Magnetic Tapes

Disks, magnetic tapes and cassette tapes have already been discussed. **Keyboard**, the essential component of any microcomputer or CRT, allows one to enter information, instructs a computer, and responds to the computer query. The keyboard of a PC is similar to typewriter. However, the configuration is not the same. Most keyboards have function keys which allow programmers to make inquiries by using one key programmed instruction to perform specific tasks.

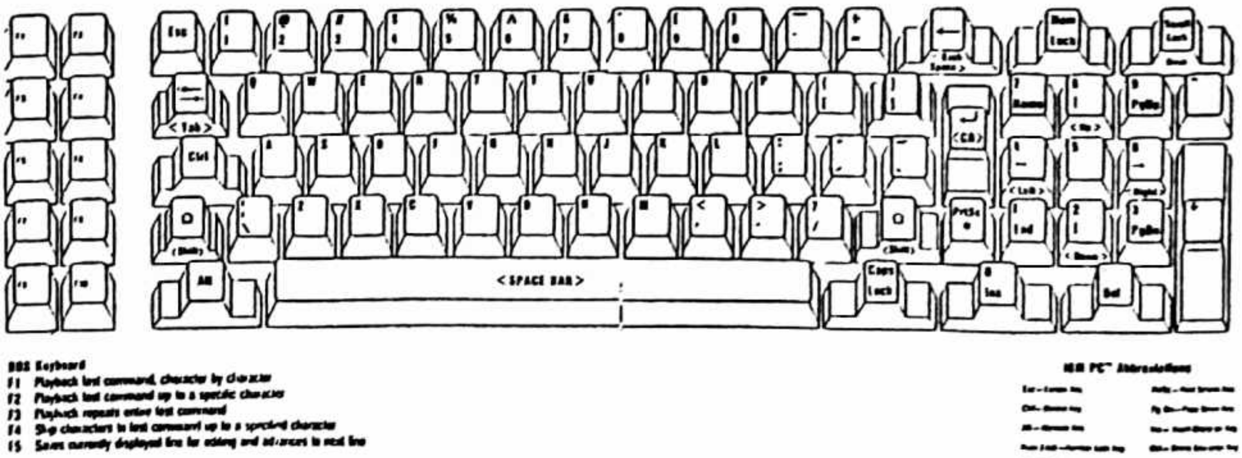


Figure 1.11 : Keyboard

The PC keyboard has three major sets of keys.

1. Function keys are positioned on the left and are labeled F1 through F10. Each software defines each function key for different purposes. Usually, the software will show the use of the function keys somewhere on the screen (monitor). The use of a function key simply requires pressing the key down. The advantage of a function is that it usually replaces a command. For example, instead of typing the command Save, one can simply use the assigned function key.

2. Numeric keys, positioned on the right, are called numeric pads and are labeled 0 through 9, . (decimal), Home, Pg Up, Pg Dn, End, Home, Del, Ins, and four arrows, one for each direction. The numeric key can be activated by holding the shift (↑) key and pressing the appropriate key or pressing the Num Lock key once and using all the upper case keys on the numeric pad. The lower case of some of the numeric keys are the arrow keys. The arrow keys are commonly used for the cursor movement. Cursor movement is the changing of the position of cursor to allow the entry of value into a location. These keys allow the movement of a cursor in four directions. Some software will not allow the use of the arrow for cursor movement. One should try them and check to see if these keys are operational.

Software

In the previous sections, computers were discussed as pieces of equipment. However, computer systems require software to run the hardware. The hardware cannot do anything without software. Software is a program which tells computers what to do and how to do it. Without software, computers are just pieces of equipment which will have no use.

Programs are sets of logical instructions which perform certain functions. Programs are classified as either application or system programs. **Application programs** perform specific functions such as payroll, personnel records, inventory record, accounts receivable, accounts payable, or other business related functions. Most of the application programs are written to help businesses in the operations of the business. Such programs are either written in-house or purchased from vendors. Most of the application programs are written in one of the high-level computer languages, such as BASIC, FORTRAN, COBOL, PL/1, or RPG. These languages use English language like rules, and, therefore, are considered easy to understand by humans. However, computers do not understand these language. Therefore, they have to be converted into the computer language by other software. This conversion software's part of the operating system's software.

System software or system program are programs which direct the internal operation of the computer. Most of system software is provided by computer hardware vendors. One of the pieces of the software is the operating system (OS). The OS consists of programs that supervise the processing of the application programs from the time it is read to the time it is completed. It instructs the computer what computer resources are needed by the program, searches for the data, and processes the output and input of the program. However, the main purpose of the OS is to reduce the processing cost by increasing the use of the various components of the computer and reducing the lost time.

The system software must include **compiler.** A compiler translates application programs into machine-readable language. That is, instructions written by the programmer, called the **source program**, must be converted into machine-readable form by compiler. The machine-readable, called the **object program**, is the program the computer uses for execution. Without a compiler, no application programs can be run on a computer unless they are already compiled by some other computer system. However, BASIC language requires either an interpreter or a compiler for converting it into a machine-readable language. Interpreters also translate

high-level computer languages one line at a time. That is, when the programmer types each line of the program, it is checked for error right away.

Programming

The purpose of a language is communication. Communication is necessary for giving instructions and passing information. Therefore, people have developed languages like Urdu, Pushto, Punjabi, Sindhi or English to make this process possible. Computer languages are developed to achieve the same objectives. Just as there are thousands of human languages in the world, there are at least twelve languages for communicating with computers. However, just as in the real world not everyone understands every one language, so also not every computer understands all computer languages yet. But unlike human beings, every computer will understand every computer language in the future.

Regardless of the computer language used, the computer is still a piece of equipment. It has to be told what to do. Unless human beings could write a computer program in which we tell the computer how to solve the problem, the computer is not capable of solving the problem. That is, we have to direct the computer what to do. Therefore, programming is a process of writing instructions in a computer language that will produce the desired results. Programming involves defining the problem, writing a sequence of instructions, and converting these instructions into computer language instructions. After writing a program in one of the computer languages, it is compiled and then run by the computer to produce the desired results. The programming process will be discussed in more detail in the next chapter.

One of the computer languages is BASIC. This book is about the BASIC language. This book discusses with examples the rules of the BASIC language. We will attempt to cover as many versions of BASIC as possible but with specific emphasis on large computers which are also adaptable to personal computers. However, keep in mind that there are minor differences among the different versions of the BASIC among

large computers as well as personal computers. Consequently, it is not possible to cover every possible variation. Therefore, you are encouraged to consult your system's BASIC Reference Manual for any unique features your system may or may not have.

QUESTIONS

1. Define CPU.

2. What is the difference between secondary storage and primary storage?

3. Is a hard disk considered primary storage or secondary storage? Discuss.

4. What is the difference between ROM and RAM?

5. What are the components of CPU?

6. Discuss the two major components of a computer system.

7. Discuss the differences among micro, mini, and mainframe computers.

8. Discuss why we need software.

CHAPTER 2

BASIC AND THE BASIC SYNTAX

INTRODUCTION

Professors John G. Kemeny and Thomas E. Kurtz of Dartmouth College developed BASIC in the mid-1960s. **BASIC** is an acronym for **Beginner's All-purpose Symbolic Instruction Code.** It was designed to allow interacting computing for on-line computer systems. Interactive computing allows programmers or users to interact with the program while it is being executed. That is, it can accept input from the user and can respond to the user. Further, a program written in BASIC is entered into the computer through either a keyboard in case of a new program or a diskette if an existing program. The compilation occurs when the programmer types the program and issues the command RUN to execute the program. It is easy to learn and can be used for a wide variety of useful business and personal applications. It has become one of the most popular programming languages. Today, BASIC is available on all types and sizes of computers.

Like any other language, BASIC has rules for spelling, syntax, grammar, and punctuation. The rules of a language help us learn a language and help us communicate with one another; so do the rules of BASIC. The rules of BASIC help us to tell the computer what we want it to do. It is interesting to note that the English language and BASIC have many words in common -- words like LET, GOTO, FOR/NEXT, INPUT, PRINT, and END link. Thus, many abstract algebraic expressions can be conveyed by easy-to-understand English words.

The main purpose of BASIC is to allow interactive use of the computer: one or more BASIC users could communicate with the large computer during the processing and feel as though they had the computer all to themselves. As the demand for microcomputers and minicomputers increased, manufacturers of such computers felt a need for simple but effective languages for them. Rather than create entirely new languages, most decided to offer BASIC because of its interactive capability -- the ability of a user to communicate directly with the computer in a conversational fashion. However, many altered the original BASIC to suit their equipment. Although BASIC has a universally accepted set of standard rules called ANSI BASIC, each manufacturer added its own "quirks" to this standard as part of the special features of its machines.

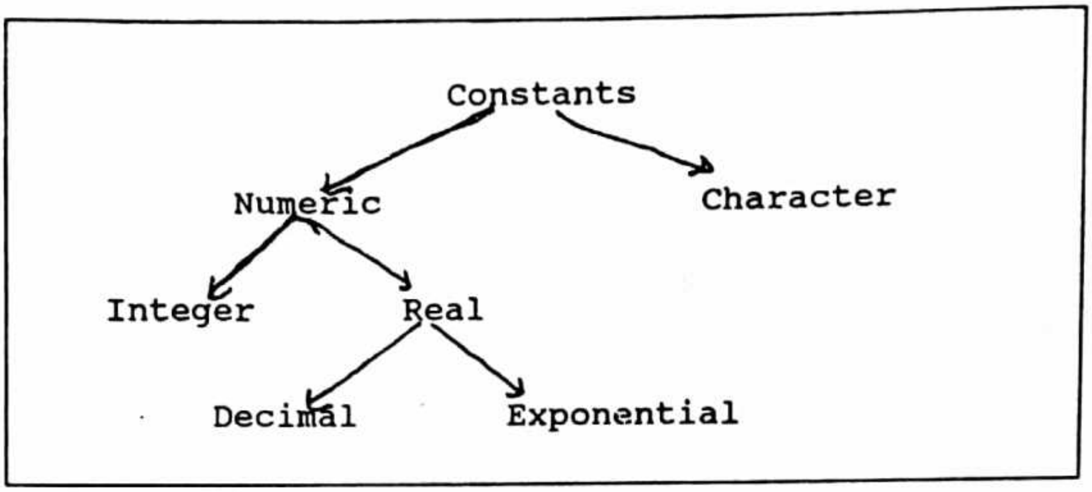
RULES of BASIC

This section defines some of the terminology frequently used by people programming in BASIC (and by computer programmers in general). A knowledge of these terms is important for a thorough understanding of this text.

Constants -- Numeric and String

Constants are values that do not change during the execution of a program. BASIC has two types of constants -- numeric constants and string constants. The following diagram illustrates the types of constants available in the BASIC programming language:

FIGURE 2.1 : Types of Constants



As depicted by the diagram above, the two major categories of constants are numbers and character strings.

Numeric

BASIC permits numbers in two ways: as <u>real</u> or <u>integer</u>. A numeric constant is any numerical value that is assigned to a numeric variable or is written as a part of an arithmetic expression. Numeric constants represent ordinary numbers can be written as decimals (called real numbers) or integers (whole numbers). Real numbers can be expressed in either exponential notation or decimal fractions. Any positive or negative whole number without a decimal is called an integer constant. For example, 65321 or 6543 or -46699 are examples of integer constants. Any positive or negative real number with a decimal is called a real (fixed point) constant. For example, -62.09 or 6365.29 or -0.35678 or 0.35678 are examples of real constants. The rules for creating integer constant or decimal constant numbers in BASIC are:

1. A number cannot have embedded commas:

96765 (valid)	96,765 (invalid)
6878.50 (valid)	6,878.50 (invalid)

2. All negative numbers must be preceded by a minus sign:

-653 (valid)	653- (invalid)
-67676.60 (valid)	67676.60- (invalid)

3. A number without a sign is assumed to be positive:

6753 is the same as + 6753
6753.50 is the same as + 6753.50

The largest number that BASIC accepts is about 1.7 x 10^38. Any acceptable number can be entered into the computer in either standard or scientific (exponential) notation. The form in which BASIC displays a number depends on many factors, with size being an important consideration. When very large or very small numbers are involved, an exponential (E) notation can be used. The decimal notation is based on a systematic use of exponents (E). Any positive or negative real number with an E within a number is called an exponential (floating point) constant. Scientific notation is commonly used to represent large numbers. Scientific notation is based on using powers of 10 to stand for zeros. In BASIC, numbers are written in the form Y x 10^i where Y is a number from 1 up to 10, and i is an integer. Y x 10 ^ i is usually written as YEi. The YEi is read as "Y times 10 to the power of i". For example, it is much more convenient to write the United States Federal Government budget 1,000,000,000,000 as 1 x 10^12 dollars, 1. E 12. It is easier to write the number 0.00000093 in E form 93 x 10^-8, or 93E-8. When a number is written in an exponential notation where the value to the left of the E is called the mantissa (significand) and the value to the right of the E is called the exponent. The computer displays i as a two-digit number, preceded by a plus sign if i is positive and a minus sign if i is negative. That is,

93. E -8

mantissa exponent

The exponent indicates the number of places the decimal should be moved to the right if the sign is positive, and to the left if the sign is negative. Note that

48780 = 4.878 x 10^ 4 = 4.878E+4 or 4.878E4 or 4.878E04

and

.007868 = 7.868 x 10 ^ -3 = 7.868E-3 or 7.868E-03

In the above examples, the decimal in the first number was moved four places to the right from its location, and in the second number the decimal is moved three places to the left of its location. Other examples of exponential notations are in given in Table 2.1.

TABLE 2.1

Examples of E-Type Constants

Decimal Number	Scientific Notation	Exponentiation Notation
653265.77	653.26566×10^{4}	65.326566E+4
.00035	$35. \times 10^{-5}$	35.E-5
.00035	3.5×10^{-4}	3.5E-4
-632.66	-6.3266×10^{2}	-6.3266E+2

To use an exponential constant, the following <u>rules</u> must be observed:

1. A number, called significand (mantissa), must precede the E. Any valid integer or decimal constant is considered an acceptable significand.

2. The exponent (number that follows the E) must be an integer constant with a positive or negative sign. If a sign is absent, a positive sign is assumed. If the exponent is too large for the computer, a diagnostic (error message) is issued. Some computer will accept a number up to 36 digits long, if a real number; and up to 18 digits it an integer. However, keep in mind that the number of digits in mantissa (significand digits) cannot exceed 16 digits for a real number and 8 digits for an integer number. Therefore, the following numbers will result in error.

Examples:

-4.567E38 or -4.567E+38

778E+20 or 778E20

9999995689E10

4.91872634E-38

In the above examples, all the numbers are larger that the permissible size.

String

A collection of alphabet, digits, or special characters enclosed within quotation marks is called a string constant. Unquoted strings are permitted only in the DATA

statements or as inputted data. A character string is a sequence of letters of the alphabet, numbers, and special characters enclosed in quotation marks. Note that the integer constant 1988 and the character constant "1988" represent two different values. A numeric constant 1988 may be used in arithmetic expressions; however, "1988" is the string value of the digits 1,9,8 and 8. The quotation marks indicate the beginning and the end of the string constant. The string constant "1988" cannot be used in arithmetic expressions. String constant can be used to represent the name of a person, thing, or place. A string constant is used for report heading and printing messages. To create a string constant, these rules must be observed:

1. A string enclosed in quotes consists of all characters between quotes, including blanks. For example:

   "ABC " , " XYZ " , " Dollar Sign ($)"

2. A null (blank) is represented by a pair of quotes. For example:

   " "

3. Any character can be used in quoted strings. For example:

   "¥:$%" , "$90" , " HELLO, How are you?"

4. The character quote(") within a quote must be enclosed by two quotes. For example, a string constant:

   "Quote ""quote"" within a quote"

Some examples of numeric and string constants are:

Constant	Type
1. 87	Numeric (integer)
2. -164.6	Numeric (decimal)
3. "What is your name?"	Character
4. "1988"	Character
5. 4.878E+4	Numeric (exponential)
6. 7.868E-3	Numeric (exponential)

Variables

A programmer can name a storage location in the CPU for storing a value in it. The storage location's name is called a variable name. In terms of computer memory, the "variable" in a program represents a specific location in the computer's memory. That is, a variable represents a place where data is stored. The value in the variable can be changed during the execution of the program; thus, the name variable. Values are assigned to variables by using BASIC statements in a computer program. In BASIC, the variable is named by combining characters, the first character must be an alphabet, however.

BASIC provides all of the functional capabilities of high-level programming languages; it has alphabet or character set, reserved words, and grammar.

The character set of the BASIC language constitutes the set of individual symbols that are recognizable by the language interpreter. The character set for BASIC varies among the systems. However, the most common character set consists of three subsets.

1. **Alphabetic characters** **A B C D E F G H I J K L M N O P Q R S T U V W X Y Z**
2. **Digits** **0 1 2 3 4 5 6 7 8 9**
3. **Special symbols** **! @ # $ % ^ & * ( ) + _ - { } [ ] : " ' ; ` , . / ? = b (b̄lank) < >.**

Certain words (such as LET, PRINT, and READ) in BASIC, called reserve words, have a special meaning to the BASIC interpreter (compiler), and thus cannot be used as variable names. Each BASIC user's manual contains a list of the reserve words for its system. Reserve words are formed with the character set. Reserve words are language commands that must not be used by a programmer to create variable names.

The programmer is responsible for naming the variables used in a program. A variable is a value that is referred to by a name in a program. Each programming language has its own rules for naming variables. The variable name in many large mainframe BASIC language must not exceed two characters in length and must consist of alphabet and digit or a dollar sign ($). On the other hand, in many PCs' versions of BASIC, variable names can be up to 40 characters in length, but only the first one to seven (in the case of the microcomputer, Microsoft recognizes 40 characters) characters are used by the computer. For example, Applesoft BASIC permits user-defined variable names to contain as many as 238 characters; however, only the first two are significant. In a similar manner, only the first two characters of a variable name are meaningful in the TRS-80 Model III BASIC. On the other hand, TRS-80 Model 4 BASIC permits names with eight significant characters. Therefore, although the variable names MASS1 and MASS2 look significantly different to a programmer, they are considered to be the same variable, MA, by the computer. The first character of all the variable names must be an alphabet character.

In IBM PC BASIC, variable names can be up to forty characters long, must begin with a letter, and may consist of letters, digits, and periods. The computer does not distinguish between upper and lower case letters used in variable name, and treats them all as uppercase. Although the letters may be typed in either case, the computer always displays variable names in uppercase letters in program listings. Some examples of variable names are TOTAL, TAX.RATE.1987, and N. The value assigned to a variable can be changed during the execution of a program. If possible, the variable names should be as descriptive as possible. That is, instead of naming the amount of loan - principal as P, it is better to use the variable name PRINCIPAL.

The BASIC language uses two data types - **numeric and string**. The data type could be either a numeric variable or a string variable. Each type of variable can hold its own type of data. In many PC versions of BASIC , the data type of a variable can be identified by the tag that is appended to its name.

Numeric Variable

A numeric variable consists of a letter or a letter followed by a digit. For example,

S, P1, S3, R9, or D

are all examples of valid numeric variables. That is, all of these names start with a letter and a digit as a second character. However, some systems differentiate between an integer and real variables. In some systems, the integer variables are tagged as % and can be assigned values between -32,768 to +32,767. Integer variables can also be assigned values without a decimal or fractions. When decimal values are assigned to integer variables, the part of the number following the decimal point is ignored. The real (a number with a decimal) data type is the default or assumed data type for the BASIC language. Any numeric variable that is not tagged as an integer will automatically be designated as a real variable. However, for a large system, the following rules apply to numeric variables:

1. Numeric variables represent only numeric data
2. Numeric variables are preset to zero prior to the program execution
3. The absolute value of a numeric variable must be in the range of 3.1352 times 10^{-294} to 1.26501 times 10^{322}.
4. Values smaller than the minimum are set to zero, and values larger than the minimum result in an error.

Numbers can be stored in a single-precision or double-precision variable. The difference between single-precision and double-precision is the number of significant digits that can be represented. Single-precision variables can be assigned values with up to seven significant digits, while double-precision variables can be assigned values with up to seventeen significant digits. The difference between the two types of variables is in the precision. A double-precision variable can store a value with 16 or 17 significant digits of precision, depending on the computer system being used. In Microsoft, the double-precision variable is tagged with the pound symbol (#). Some BASICs cannot use double-precision. Many non-scientific applications use the integer and single-precision data types.

String Variables

String variables represent alphanumeric text and are named with a 2- or 3- character identifier. String variable names are formed by adding the $ (dollar sign) tag to a variable name. The first character must be an alphabet. For example,

A$, C$, N$, or Y$

are examples of string variables.

Values that can be assigned to string variables are termed string values or simply strings. A string is any combination of the BASIC character set. For example, "sentence" is a string, and "99999999" is also a string. The latter example appears to be a number, but it is considered a string by the computer.

A string can be created from any combination of elements of the BASIC character set; therefore, some numbers could be read as string. However, the context within which these "numbers" are used makes them strings. In some cases, it may be desirable to read a number as a character string, e.g. a social security number (SSN) or an ID number. This

type of number is not likely to be used in arithmetic operations, and, therefore, it does not make sense to treat it as a number. It is important to remember that strings cannot be assigned to numeric variables and string values cannot be used in arithmetic operations.

Some examples of valid variables are:

Type of System	Variable	Type
Mainframe:		
	Y1	integer/real
	Y	integer/real
	A1	integer/real
	A	integer/real
	B$	string
PC		
	SUM%	integer
	TOTAL	real
	NAME$	string
	A	real
	A%	integer
	A$	string

Some examples of invalid variables are:

Type of System	Variable	Reasons for error
Mainframe:		
	1Y	First character is a digit
	Y?	Second character (?) is a special character
	A-	Second character (-) is a special character
	$A	First character ($) is a special character
PC		
	%SUM	First character (%) is a special character integer
	1TOTAL	First character is a digit
	$NAME	First character is a special character

STATEMENTS

In applied problems, quantities are referred to by names. In general, a variable is a name that is used to refer to data. LET statements are used to assign values to variables or write mathematical expressions and PRINT statements are used to display the values of variables. Each basic statement is preceded by a number which identifies the statement as well as its sequential relationship to the other statements in the program.

The LET Statement

In some systems, the assignment of a variable name requires the use of the LET statement. The general form of the LET statement is

```
Ln     LET   varname = constant or arithmetic expression
```

The LET statement consists of the line number (LN), the word LET, the = (equal sign) and a constant (numeric or string) or an arithmetic expression.

The LET statement assigns a value to a variable or another variable to a variable name. For example,

```
10    LET   varname = "small"         (a string constant)
20    LET   varname = varname         (another variable)
30    LET   varname = 99950           (a numeric constant)
```

Let us assign the numeric constant 568 to the integer variable SUM%; that is, the statement

```
10  LET SUM% = 568   or   10 LET A1 = 568
```

assigns the value 568 to the variable SUM% or A1. Actually, the computer sets aside a location in memory with the name SUM% or A1 and places the number 568 in it. The statement

```
20    PRINT SUM%    or 20 PRINT A1
```

looks into the SUM% or A1 memory location for the value and displays it on the screen.

Variables are commonly used to represent an unknown. For example, the number of inches (I) can be represented as follows:

```
10 LET   I = 10 * 12

            or

10 LET   F = 10
20 LET   I = 12 x F
```

That is, every time the number of feet (F) increases by 1, the number of inches increases by 12. Since the number of feet (F) and the resulting number of inches (I) can vary, F and I are called variables.

Consider the following relationship which can be used to compute the weekly or monthly pay for a person employed at an hourly rate:

```
10   LET   A = R1 * H1
```

All the variables are unknown to the computer. Therefore, values must be assigned to the variables R1 and H1 before this statement is executed. For example, a complete

program could look like as follows:

```
10      LET      R1      =      10
20      LET      H1      =      20
30    LET    A          =    H1    *    R1
40 PRINT R1, H1, A
```

As the value of either R1 or H1 varies, so does the weekly pay represented by the variable A (AMOUNT).

ASSIGNMENT RULE
Every variable in the LET statement to the right of the equal sign should have appeared in the program previously so that its value is determined.

Consider a savings plan where interest is compounded annually. In this situation, starting with a principal of P dollars earning i percent interest per year, the resulting amount (A) after n years can be written as follows:

```
10 LET  A = P * (1+i) ^n  or    10  LET A = P * (1+i) ** n
```

A different value of the variable P, i or n will result in a different total amount of dollars represented by the variable A.

Expression

Some statements of BASIC are called expressions. An expression is formed by combining operands and operations; however, an expression could be formed with just constants or variables. Expressions can be arithmetic or logical. An arithmetic expression consists of a sequence of variables, constants and arithmetic operators that is to be evaluated to calculate a number. Expressions are evaluated by replacing each variable with its value and carrying out the arithmetic operations. Arithmetic and logical expressions are used frequently to solve problems.

Arithmetic Expression

Arithmetic operators are used to indicate operations like addition, subtraction, multiplication, division, and exponentiation:

TABLE 2.2

Arithmetic Operators

Operation	Priority of Operations	Mathematical symbol	BASIC Symbol	BASIC Example
Addition	4	+	+	C+Y+9
Subtraction	4	-	-	D-9-4
Multiplication	3	x	*	Q*T*E
Division	3	—	/	Q/5
Exponentiation	2	C^D	** or ^	C^D or C**D
Parentheses	1	( )	( )	(A+B)*C

In the case of exponentiation, the symbols "^" or "**" could be used. The user's manual will identify the acceptable symbol. However, some versions of BASIC use both "**" and "^" for exponentiation.

The following rules describe the order in which the operations in an arithmetic expression are evaluated:

Precedence Rule 1: If there are no parentheses in the numeric expression, first all exponentiations are evaluated, then all multiplications and/or divisions, and finally all additions and/or subtractions. However, in case two of the same priority operators are present in the equation, the one on the left is evaluated first. That is, in case the division and multiplication symbols are present, the one on the left will be evaluated first. The same is true of the plus and minus.

This order of operations is sometimes called the rules of precedence or the hierarchy of operations. The meaning of these rules can be made clear with some examples.

For example, the expression

20/5*3/3^3+6*5-6 (2.1)

is evaluated as follows:

20/5*3/9+6*5-6	operation 1 (3 to the power of 3)
4 * 3/9+6*5-6	operation 2 (20 divided by 5)
12/9+6*5-6	operation 3 (4 multiplied by 3)
1.333 + 6* 5-6	operation 4 (12 divided by 9)
1.333 + 30-6	operation 5 (5 multiplied 6)
31.333 - 6	operation 6 (1.333 plus 30)
25.333	operation 7 (31.333 minus 6)

Parentheses are used to change the order of operations. The following rules describe the order in which the operations in an expression containing parentheses are evaluated:

Precedence Rule 2: If there are parentheses in an expression, all parts of the expression within the parentheses are evaluated first according to the Precedence Rule 1. Then the remaining expression is evaluated according to Precedence Rule 1.

Using the first example (2.1) with parentheses, the expression:

```
20/5*(3/3)^3+6*5-6        (2.2)
```

is evaluated as follows:

20/5*1^3+6*5-6	operation 1	(3 divided by 3)
20/5*1+6*5-6	operation 2	(1 to the power of 3)
4 * 1 +6*5-6	operation 4	(20 divided by 5)
4 +6*5-6	operation 5	(4 multiplied by 1)
4 + 30-6	operation 6	(6 multiplied by 5)
34 - 6	operation 7	(4 plus 30)
28	operation 8	(34 minus 6)

Calculating interest on a loan is an example of an arithmetic expression. Some other examples of arithmetic expressions are:

```
B =  22 * RATE  + 7

C =  M + 1

D =  (C + D) /3
```

If D is a variable, then the statement

```
LET D = A + B
```

first evaluates the expression on the right and then assigns its value to D. Because the expression on the right of the equal sign in a LET statement is evaluated before an assignment is made, a statement such as

```
LET AMOUNT = RATE * PRINCIPAL
```

is meaningful. It first evaluates the expression on the right and then assigns the value to the variable AMOUNT. The effect is to calculate the value of the variable AMOUNT by multiplying RATE by PRINCIPAL. In terms of memory locations, the statement reads "the values of RATE and PRINCIPAL" from the RATE's and PRINCIPAL's memory locations respectively, uses them to calculate the AMOUNT, and then places the number in the AMOUNT's memory location. Other examples are:

```
10  LET  Y  = 10
20  LET  X1 =  5
30  LET  A1 = 10
40  LET  A  = Y+333
50  LET  B  = X1+Y-556
60  LET  C  = A1*B/3222
70  LET  D  = 5.6*C*Y
80  PRINT A,B,C,D
```


To use an arithmetic expression, the following rules must be observed:

1. Only numeric operands (constant or variable) and numeric operators should be used.

2. Two arithmetic operators cannot appear side by side:

Algebraic Expression	Valid BASIC Expression	Invalid BASIC Expression
X * - Y	X * (-Y)	X * -Y
X / - Y	X / (-Y)	X / -Y

3. Operators cannot be implied:

Y/X(A*B) (invalid)   Y/X * (A*B) (valid)

Logical Expression

A logical expression is a sequence of variables and constants, connected by relational operators. Logical expressions always have a value of true or false, test the condition, and indicate whether to take alternative courses of action based on the results of the tests. For example:

```
C > D
A + B <= 0
A < D AND D > F
```

If the above conditions are written in a logical statement, it will be checked to find out whether a condition is true or false. The comparison can also be made between two numerical values or two string values. In a logical expression, any two constants or variables are related by using relational operators.

There are two types of logical expressions: simple and compound. Simple expression are formed by connecting two numeric or string expressions with a relational operator. Compound relational expressions are formed by connecting two simple relational expressions with a logical operator.

Relational Operators

A number of rational operators are used by BASIC for logical expressions. Logical expressions, such as A>B, have a value of either true or false and can be used to establish the condition under which the task is to be performed in a program. For example, using the logical expression A>B, one can cause a branch to a specific set of statements in a program whenever this expression is true, i.e. whenever the value of A is greater than the value of B. The relational operators are:

TABLE 2.3

Relational Operators

Relations	Mathematical Symbol	BASIC Symbol	Examples
Equal to	=	=	Y = X
Less than	<	<	A > 50
Greater than	>	>	X1 > 9
Less than or equal to		<=	Y2 <= 20
Greater than or equal		>=	Y4 >= 50
Not equal to	=	<>	A <> B

A condition test is coded within an IF statement. The IF statement will be discussed under IF ... THEN statements. The condition in an IF statement could consist of compound conditions. That is, conditions that are separated by OR, NOT, and/or AND are called compound statements. For example,

A>B and C>B; it will perform the condition if both values of the variables A and C are greater than B. However, if A is less than B, then the condition C greater than B will not be checked, and the statement will not be executed.

A>B or C>B; it will perform the condition if either A is greater than B or C is greater than B

A>B or C>B and D=E; it will check the first two conditions, and if one of them is true, it will only then check the D=E condition. otherwise, the last condition will not be checked

NOT (A>B); this statement is true only if the statement A>B is true. In other words, A is less than B.

QUESTIONS

1. Define a string and a numeric constant.

2. Define an integer and a floating point constant.

3. Define a variable. Discuss with examples string and numeric variables.

4. Classify the following numbers as either an integer or a floating point.

   45000 650.40 560.00 6000 6600 .689 450 88 8.8

5. Find errors, if any, in the following data, and correct them if not valid constants.

   45,000 450*40 450 45.00 450/50

6. Which of the following are valid integer numbers. Correct those which are not valid integers.

   45.00- 45.00 +4506 56,000 .60 -60 60.90

7. Which of the following are valid floating point numbers, correct those which are not valid floating numbers.

   45.00 450.00- 4506 +45.00 .60 60 60.90

8. List the symbols and the character sets used in BASIC.

9. Find errors, if any, in the following variable names. Correct them and explain the reasons for the errors.

   S1 S-2 #S1 2S S2# S$ S1$

10. Which of the following are valid string constants? Correct those which are invalid.

"KARACHI" TEST "456" "456,78" "PAKISTAN" 456

"517-773-000" LAHORE,PAKISTAN

11. Which of the following are valid BASIC statements? Correct them, if wrong.

```
a) 10 LET A = 10
b) 10 LET 10 = A
c) 10 LET A = 10 = 20
d) 10 LET A = "KARACHI"
e) 10 LET A = KARACHI
f) 10 LET A$ = "KARACHI"
g) 10 LET KARACHI = A
h) 10 LET A = B/10 + T*10 (assume that B and T are defined)
i) 10 LET B/10 + T*10  = A
j) 10 LET A = 3 + A$ /34
l) 10 LET A$ = 3 + 45 /5
```

12. Which of the following are valid expressions?

```
A =  45 / 4 * 10 + 14
B =  A / 10.10 - - 45 / 10
C = -A / 10.10 + 45
D =  A / -10.10 + 50
C = 56 * 10 / 56 + 14
C = (56 * 10 / 56 + 14 ) )
```

13. Calculate the answers to the following statements.

```
A = 45 / 50 + 450 /10 - 60 * 3 + 3 ** 2
A = (45 / 50 ) + 450 /10 - (60 * 3) + 3 ** 2
A = (45 / 50 ) + (450 /10) - (60 * 3) + 3 ** 2
A = ((45 / 50 ) + (450 /10) - (60 * 3)) + 3 ** 2
```

14. **Write the exponential equivalents of the following numbers.**

a) 459.34
b) - 45.456
c) 2345.56
d) 0.000456
e) 40560.

15. **Write the decimal equivalents of the following numbers.**

a) 6.788E+04
b) .6788E+05
c) 6.788E-04
d) 5.629E-01
e) 5.68 E+02

16. **Which of these numbers are invalid exponentials.**

a) 6.78E04
b) 6.56E
c) .56-02
d) 6780.
e) 567.56E-

17. **Define a logical expression.**

18. **Define relational operators.**

19. **Write the following algebraic expressions as computer expressi**

a) $Y^{2} + XY + 2/99$
b) $X^{2}\ Y^{2} + (Y + X)^{2} / X$
c) $(6 - X / 9 + Y)^{2} / (A + B)\ (CD)$

20. **Assume that the value of X is 3, C is 5, and Y is 4. Evaluate the following expressions.**

a) Y + X - C
b) X/Y + Y ** 2 + C
c) (X/Y + Y) ** 2 + C
d) X + Y - C
e) X/Y + Y/X + C
f) X/Y + (Y/X + C)

CHAPTER 3

COMPUTER PROGRAMMING PROCEDURES

COMPUTER PROGRAMS

Computer programs (software) are step-by-step instructions for solving a problem. The computer must be able to read and interpret each of these instructions. Writing a program requires the following standard procedure called "the programming process". That is, one must:

1. Define the problem -- understand what one is trying to solve.
2. Design the program -- write a step-by-step procedure (called algorithm) for solving the problem.
3. Code the program -- write the program using the language of one's choice, e.g. BASIC.
4. Type the program -- enter the program into the computer.
5. Execute and Debug the program -- run the program and correct syntax error, if any.
6. Revise and execute the program -- fix all the errors and run the program with test data until an acceptable output is produced.

Let us take a problem for the purpose of showing the programming process. For example, one wants to borrow $10,000 to be paid in equal installments over the life of the loan for 15 years at 10% interest rate. The question is: how much is the interest per year, and how much will be the total interest during the life of the loan?

Define the Problem

The first step is to define the problem. The problem could be analyzed by using the basic flowchart -- input, processing, output -- of the data processing. Before determining the algorithm for solving this problem, one must:

1. Determine the desired form of the output. For example, does one want the amount of the interest of a particular year? Does one want a table showing a yearly balance, the amount of interest each year, and the total amount of interest after each year? Or does one want just a column showing the amount of interest for each year?

2. Determine how the data will be inputted. For example, what data is needed? What is the type (numbers or strings) of data? What is the source of data (terminal, tape, or disk)? Is there a need for error checking on the data?

Design the Program

After deciding the above parts of the program, the second and the most important and difficult step in defining the problem is the ability of the programmer to logically organize the steps of the program. This step requires the knowledge of formulas or data manipulation methods to convert the input into the output. For example, the calculation of the interest amount requires the following formula:

AMOUNT = PRINCIPAL x INTEREST RATE (3.1)

This is the algorithm to calculate the interest amount. Therefore, we know how to solve the problem. Of course, the computer has to be told how to solve a problem. Therefore, if one does not know how to solve a problem, the computer does not either. Furthermore, if one has to calculate the total amount of interest paid at the end of each year, the algorithm as is follow:

SUM = SUM + AMOUNT (3.2)

This means that the variable SUM is accumulating the amount of interest (AMOUNT) paid each year.

Code the Program

The third step -- writing and documenting the program -- should be relatively easy. Programs are written by translating an algorithm into a series of language statements. The programming languages are similar to sentences written in English, a natural languages. Many BASIC words such as READ and PRINT are easy to interpret. The language has precise grammatical rules. Further, the programmers must develop an effective and efficient writing style like the one required for English.

Grammar is generally referred to as syntax in a programming language.

A BASIC program consists of a series of statements that tells the computer what to do. Each statement consists of a programming command with numeric or character string constants or numeric or string variables, or formulas called expressions.

The command code of the BASIC instruction determines the function. The command determines the processing task to be performed by the statement. For example, if the BASIC command LET was a part of a statement, a variable memory will be created and a value will be stored in the memory. Similarly, the PRINT command starts an output operation. Essentially, all statements must contain commands or an expression. Otherwise, the statement will have no purpose.

All statements with command must have a variable or a constant to be worked on. For example:

```
READ   N
```

The statement defines the variable N to be processed. This statement specifies the operation of READ and specifies to read the variable N.

Statements written in a BASIC program follow the general line format consisting of three elements: (1) line number, (2) command, and (3) variable(s). Each component in this format serves a different function. For example, a program for the equations (3.1) and (3.2) in BASIC would look as follows:

PROGRAM 3.1

```
10  LET P= 10000
20  LET I=.15
30  LET S= 0
40  LET A = P * I
50  LET S = S + A
```

The line numbers in a BASIC statement identify the statement and define the order in which the instructions are executed. The line number starts each instruction and assumes a numeric, integer format. The range of acceptable line numbers is generally between 1 and 99999. The upper limit depends upon the systems for which the BASIC program is written. Generally, the line number 99999 could be the last statement number possible, though it is very rarely used. Typically, spacing between lines is suggested when numbering lines so that additional programming statements can be inserted, if necessary. Line numbering such as 11, 12, 13, 14, 15, does not allow for changes or additions to the program. Therefore, line numbering such as 10, 20, 30, 40, 50, is advisable. These numbers allow for additions between existing lines, if desired. For example, a line

number 32 can be inserted between the above lines 30 and 40. None of the existing statements has to be renumbered. The operational software supporting BASIC will accept this newly inputted statement and logically position it between lines 30 and 40. Thus, the new order of statements becomes, for example, lines 10, 20, 30, 32, 40, and 50.

No two statements could have the same line number. Some versions of BASIC allow several statements to be placed on a single line separated by colons. That is, more than one statement can be placed on a single line of a program provided that the statements are separated by colons. For example, to rewrite Program 3.1 in a multiple form,

PROGRAM 3.2

```
10  LET P= 10000
20  LET I=.15
30  LET S= 0
40  LET A = P * I : Let S = S + A
```

This practice, however, it is not advisable because it increases the difficulty of writing, reading, documenting, debugging, and revising programs. Therefore, the general rule of just one statement per line is recommended. This rule also makes editing easier.

The line numbering method gives BASIC a true advantage. Users are free to enter BASIC instructions in any order, and BASIC will properly reorder them prior to their execution. This facility makes corrections of a program easier, thus simplifying the debugging process.

In each statement, a reserved word or an expression should be placed after each line number. A reserved word defines the task to be performed. An expression is used by a

reserved word in the process of manipulating data or making decisions. Expressions include values to be assigned to variables.

The place of the statements within a program determines how the program is executed. However, it is advisable to follow a standard structure in the development of all programs. The organization of a program is referred to as programming style.

Programming Style

One of the preferred programming styles is to create a program that includes a section called **documentation.** That is, this section should identify the purpose of the program, author, date written, date revised, variable names, and variable usage information. The REM (remark) statement is used to document a program. The REM statement has no effect on the program. The remark statement contains comment or explanation intended for humans. For example,

```
1 REM    PROGRAMMER    JOHN DOE
2 REM    This program calculates interest on a loan
5 REM    VARIABLES
6 REM     A = AMOUNT    P = PRINCIPAL    I=  INTEREST
7 REM     S = SUM
```

Following this section, the declarative statements, if any, should appear. **Declarative statements** define arrays, data sets, and data types (Examples of these will be discussed later in the book). The third section should consist of the program body or data manipulation portion of a program. The body of the program contains a section on input, process, and output. In addition, liberal documentation of statements within the program is highly recommended. The apostrophe (') or an exclamation (!) could be used for documentation within the executable statement. **Executable statements** are statements which the computer checks and

processes. For example, to rewrite statement 40 in (3.1),

```
10   LET   A = P *  I     'This statement calculates interest
```

In this example, the statement before the apostrophe (') will be executed, and the message after the apostrophe will be ignored. Documentation is always helpful to a reader of the program. It provides a record of the intent of the various programming sections. Good documentation can help avoid future problems or make them easier to solve when they occur. Documentation consists of all written descriptions and explanations necessary for later modification or updating of the program. The last section of a program is program termination. That is, the END statement. For example,

```
120  END
```

Type the Program

To begin entering a new program, type each line as follows and press CR (carriage return), RETURN, or <- key after each statement. For example:

PROGRAM 3.3

```
 10   REM    PROGRAMMER   JOHN DOE                              (CR)
 20   REM    This program calculates interest on a loan         (CR)
 30   REM    VARIABLES                                          (CR)
 40   REM    A = AMOUNT   P = PRINCIPAL    I=   INTEREST        (CR)
 50   REM    S = SUM                                            (CR)
 60   LET    S = 0                                              (CR)
 70   LET    P = 10000                                          (CR)
 80   LET    I = .15                                            (CR)
 90   LET    A = P * I                                          (CR)
100   LET    S = S + A                                          (CR)
110   PRINT   A, P, S                                           (CR)
120   END                                                       (CR)
```

However, some BASIC compilers will let the computer give line numbers. For example, IBM PC will generate line numbers by you typing AUTO command without a line number before entering the programming statements.

Execute and Debug the Program

The programmer writes these instructions, or processing steps, in a high-level computer language (BASIC), so that they are easy for humans to understand. However, the computer cannot understand any high-level language. The computer requires machine language for executing the instructions. Most of the microcomputer systems require an interpreter or compiler to translate. The high-level languages are translated into machine language by an interpreter or a compiler -- that is, programs which are stored in the computer memory, or which have to be stored in the computer memory before a higher-level language can be run. Most microcomputers use an interpreter for BASIC, while the larger systems often use a compiler for translation. This is due to the fact that a compiler typically requires much more storage space than an interpreter. In addition, a compiler creates an object program from the source program and then loads the object program into memory. The computer then executes the object program. However, an interpreter passes one line at a time of machine language to the computer. A compiled program (object program) normally executes faster than those created with an interpreter.

For execution and debugging a program interactively, most of the systems require the command RUN. Therefore, to execute a BASIC program, one types RUN and presses RETURN.

During the execution of the program, the BASIC compiler checks the syntax rules and highlights errors. The method for displaying these errors depends upon the hardware being used and the processing mode. In most cases, the BASIC program errors will be flagged as they occur, because most BASIC programs are run in a time-sharing, interactive mode.

Revise and Execute the Program

If there are syntax errors in the program, the computer will indicate the line number where the errors occurred or will stop at the line of error occurrence. In either case, one reads the statement and fixes the error either by retyping a new statement or editing the old line. After fixing errors, one types RUN again to execute the program. However, after fixing syntax errors, the program may not run; or, if it runs, it may not produce correct results. This may be the result of logical errors caused by incorrect initialization of a variable, a misplaced GOTO statement, or other out of sequence statements in the program. Therefore, it is very important that one carefully think through the programming logic at the time of writing the program. The detection of logical errors can be learned through experience, however.

In order to scan the whole or partial program that is currently in the computer memory, one just types

```
LIST        or    LIST   10-100
```

The command LIST will list the whole program; while the command LIST 10-100 will list line numbers 10 to 100. However, some systems may require a different format for listing a partial program. Therefore, one should check the system's manual for a proper rule.

Saving the Program

If the program is to be used in the future, one types the command SAVE followed by a name for the program. One should check the system's manual for the proper rule. Usually the program name can be up to six to eight characters and must start with an alphabet character. For example, to save Program 3.1, one types

```
SAVE   "interest"
```

After properly saving the program, one can retrieve it either by the command LOAD (for a personal computer), GET, or OLD. You should check your system's manual for an appropriate command.

QUESTIONS

1. List the steps needed to write a program for a problem.

2. You are planning to buy a car on credit. In order to know your ability to pay back the loan, it will be better to know the monthly interest and the principal you have to pay for each interest rate and the amount you need to borrow. Discuss the procedure you would follow for writing a program for the problem.

3. Why is it important to follow the programming procedure for writing programs?

4. Is it possible to write a program without properly defining the problem?

5. What is documentation and why should we include documentation in a program? Discuss.

6. Differentiate between executable and non-executable statements.

7. Write statements (computer commands) for listing the following lines in a program.

   a) List line number 100.
   b) List line numbers between 110 and 200, including 110 and 200.
   c) List the whole program.
   d) List line numbers between 100 and 200 and then 230 and 300.

8. Write a statement (computer commands) to save a program called "Depreciation."

9. Why do you need line numbers in a program written in BASIC language?

10. What would happen if you were to type a BASIC statement without a line number?

11. What would happen if you were to type two BASIC statements with the same line number?

12. Correct errors, if any, in the following program (do not change the statements).

```
10 LET A = 10
20 LET B = 20
20 LET C = A + B
30 PRINT A, B, C
23 READ A, B, C
25 PRINT A, B, C
14 END
```

13. What is the REMARK statement used for? Discuss by giving some examples.

CHAPTER 4

DATA INPUT – INPUT and READ/DATA STATEMENTS

DATA VALIDITY

Logic errors in a computer program can result in incorrect results. Besides the logic error, another more common reason might be that the program is given incorrect data to process. This type of error is sometimes referred as "Garbage in, garbage out (GIGO)." There are some ways BASIC language can be programmed to guard against the problem of incorrect data. That is, the program should verify data at the time it is read or inputted. Following are some of the ways BASIC can be programmed to check for invalid data:

1. Not enough data: Suppose one value is entered and then the <return> is pressed. If the computer is expecting two values, it will ask for more or ask to "redo from start." While reading data, if not enough data is available,"out of data" message is printed and the execution of the program will terminate.

2. More than enough data: Suppose one entered three values and the computer expected two values. Most computers will ignore the excess data and continue the execution without any message, while some computers will respond with EXCESS IGNORED or redo from start. In the case of the reading of data, it will not check beyond what is needed for the read statement.

3. Incorrect data type: If the computer expected a number and an alphabet or alphanumeric character(s) was entered, the computer will ask to REDO the entry. During reading data, the computer will give back a message and identify the error and the possible line number where the error occurred.

These are some of the ways the computer checks on errors. However, if these conditions do not occur, the computer will continue to operate without giving a message. There are other cases where errors can occur. In such cases, it is the responsibility of the programmer to provide for a validity check of the data. For example, if a variable requires a date, the month must be between 1 and 12 and the day between 1 and 31. Therefore, a portion of a program can be created to check that the date entered is consistent with this rule. In addition to providing for a validity check of the data, it is absolutely necessary that programs should be designed so that data entry is as convenient as possible. This will reduce the amount of mistakes. One of the ways for convenient data input is to limit the number of keystrokes a person has to make during data entry.

In BASIC programming there are three methods for accepting the data into the computer for processing. One is the LET statement, discussed previously; it is used primarily for the assignment of a single or one-time data value. The other is the READ/DATA statement combination commonly used for processing multiple records with constant values. The third method of data entry in BASIC is the INPUT statement.

INPUT STATEMENT

One of the most common methods for entering data into a BASIC program is through the computer terminal keyboard. In this type of data entry, the computer program prompts the user to enter data via the computer terminal keyboard. This type of data entry, commonly called the interactive data

entry, is common in many business data processing applications. Consider an order entry system. The salesperson asks a customer a set of questions and enters the responses into the computer through its keyboard. In most cases, questions on the computer screen are generated by a program. The questions expect certain responses and expect appropriate data type entry. The program will then retrieve information or calculate additional information based on the logic of the program and the information provided.

Interactive data entry is essential in those cases where the values of the data items are unknown until the program is actually used. In the order entry system, the salesperson does not know the customer's name and address, the item(s), and the number of units wanted. As a result, the programmer has to create an interactive data entry program. Therefore, the INPUT statement is an appropriate approach. It accepts the data from the keyboard and causes the program to pause and wait until the appropriate type and amount of data is entered by the programmer or the terminal operator.

In BASIC language, interactive dialogues between com puter and user are easy to program. The INPUT statement is used for this purpose.

The basic form of the input statement is

```
Ln  INPUT  variables separated by commas
```

The input statement is preceded by a statement number, a blank, and a blank following the keyword INPUT; and it is then followed by variable name(s) appropriate to the type of information to be inputted. The variable names are separated by a comma.

INPUT RULES:

1. The input statement must come before the statement where the variable is used in a program and whose value to be obtained through an INPUT statement from the keyboard

2. Only commas should be used to separate variables

3. No comma should be placed at the end of the INPUT statement in the program.

4. While inputting the data, the data must be separated by a comma.

An example of a correct INPUT statement is

PROGRAM 4.1

```
100  INPUT A$,I,K1
110  PRINT A$,I,K1
120  END
```

Variable names are names assigned to computer storage locations which hold the data entered through the INPUT statement. The computer must know in advance what type of data (either alphabetic or numeric) is to be stored, and this information is conveyed by the particular form of the variable names.

In Program 4.1, the data to be entered are for A$, I, and K1. As you know, A$ is an alphabetic data variable, and I and K are numeric variables. After the three values are entered by the programmer and the <return> or ENTER key is pressed, the values are assigned to the variables. As in this statement, more than one variable may appear in the INPUT statement. In such a case, commas must separate variables. These commas, however, serve only to separate the variables; they have no effect on the way the data is entered.

In Program 4.1, the execution of the program continues until the computer encounters the INPUT statement, pauses and prints a question mark (?) on the screen. After printing the question mark, the computer waits for the three data values for variables A$, I, and K1 separated by commas to be entered from the keyboard. The data values must match the type of variables in the statement. That is, the first data value must be an alphanumeric or alphabetic character(s), and the second and the third data values must be numeric values. When the program encounters the input and the print statements , the screen will look as follows:

OUTPUT 4.1

```
? SHAH, 450, 50
SHAH            450       50
```

In the above output, when the computer encounters statement number 100, it will type

?

You will then type SHAH, 450, 50 to match the variables list in the INPUT statement. After your press ENTER key, the computer prints SHAH 450 50. However, if you respond to the above question mark by only typing SHAH and pressing ENTER, the computer will respond with

```
?    Redo    from    start
?
```

This indicates that not enough data has been entered. Further, if you enter a string (alphabetic) data in response to a numeric variable, the computer will also ask for "redo from start," and wait for you to repeat the INPUT operation.

On some systems, when inputting alphabetic or alphanumeric variables, the information should be enclosed in quotation marks for each variable. Most often, on the screen a small blinking light next to the question mark will appear. This blinking light is called the cursor. As a character of the data is typed, it moves the cursor one space and the character is displayed on the screen. This is called echoing. After the data is typed and the ENTER key is pressed, the program will resume its processing until another INPUT is encountered, at which time it will again pause and print ? on the screen to request data.

However, it is advisable that whenever an INPUT statement is used in a program, the person at the keyboard should be told what type of value is being requested. That is, the person should be prompted. One of the ways to prompt a person is to have a PRINT statement preceding every INPUT. For example,

PROGRAM 4.2

```
100PRINT"Type the name of the customer,item number and quantity"
110INPUT A$,I,K1
120PRINT A$,i,K1
130END
```

Line number 100 prints the message, and line number 110 waits for the information to be inputted. The screen for this program appears as follows:

OUTPUT 4.2

```
Type the name of the customer, item number and quantity
? SHAH,450,50
SHAH                450            50
```

The PRINT statements identify the type of information the user should enter. Instead of all three values being entered on one line, the three values may be entered separately, as in the following example.

PROGRAM 4.3

```
100 PRINT "Type the name of the customer"
110  INPUT  A$
120  PRINT  "Type the item number"
130  INPUT  I
140  PRINT  "Type quantity"
150  INPUT K1
160  END
```

The screen for this program appears as follows. The small letters are messages printed by the print statements, and the capital letters are the responses typed by the user.

```
OUTPUT 4.3

Type the name of the customer
? SHAH
Type the item number
? 450
Type the quantity
? 50
```

However, if one wants the data to be entered on the same line where the message was printed, then a semicolon should be typed at the end of each PRINT line. For example, the above program would appear as follows:

```
PROGRAM 4.4

100 PRINT "Type the name of the customer";
110  INPUT  A$
120  PRINT  "Type the item number";
130  INPUT  I
140  PRINT  "Type quantity";
150  INPUT K1
160  END
```

The screen of the above program appears as follows. The messages of the program are printed in small letters, and the user has typed-in the responses in capital letters at the end of each message line.

```
OUTPUT 4.4

Type the name of the customer? SHAH
Type the item number? 450
Type the quantity? 50
```

Because the BASIC language is meant for interactive programming, prompting for inputting data has been made a part of most versions of BASIC. That is, INPUT statements can include a message within the statement. For example, the above program could be entered as follows:

PROGRAM 4.5

```
100  INPUT "Type the name of the customer",A$
110  INPUT "Type the item number",I
120  PRINT  "Type quantity",K1
130  END
```

The screen (output) for the above program appears as follows. The program has printed the messages in small letters and the user has typed the responses in capital letters.

OUTPUT 4.5

```
Type the name of the customer? SHAH
Type the item number? 450
Type the quantity? 50
```

Suppose you want to create a program where customers are billed based on the type of item and the quantity they buy. The following program shows how to input the data and print out the bill.

PROGRAM 4.6

```
100 PRINT "Enter the customer's name";
110 INPUT N$
120 PRINT "Enter item number";
130 INPUT I
140 PRINT "Enter quantity";
150 PRINT K1
160 PRINT "Enter price";
170 INPUT P
180 T= P*K1
190 PRINT  N$
200 PRINT "Quantity ", K1, " Price per unit $", P
210 PRINT "Total Amount $ ", T
220 END
```

In this program four values for variables, N$, I, K1, and P will be needed, and the computer will print out the customer's name, item number, quantity ordered, the price per unit, and the total amount. The input and the output will look as follows:

OUTPUT 4.6

```
Enter the customer's name? SHAH
Enter item number?  211
Enter quantity? 10
Enter price? 5

 SHAH
 Quantity       10    Price per unit $           5
 Total Amount $       50
```

READ STATEMENT

The INPUT statement is one way of getting data values into a program. When one uses the INPUT statement, the computer types a ? and then waits for a user to type in a value. After one types it in and presses RETURN or <enter> key, the computer then stores that number in its memory. But if one has a lot of data that won't change from run to run, then the READ/DATA statement is a better method for getting information into the computer. The READ and DATA statements provide another way to enter data into a BASIC program. These two statements always work together. This method of entering data into a program works a little differently than the INPUT statement. The READ statement searches through the BASIC program until it finds the first DATA statement. The computer then assigns the data values in the DATA statement consecutively to the variables in the READ variable list. That is, values contained in the DATA statements are assigned to variables listed in the READ statements.

The READ statement reads data from a sequence of data items in the DATA statement and assigns them to the variables in the READ statement. The READ statement consists of the keyword READ followed by a list of variables separated by mandatory commas. The variables may be numeric or string variables. The general format of the READ and DATA statements are as follows:

```
Ln   READ variable list
Ln   DATA data list
```

Each READ or DATA statement is preceded by a statement number, a blank, the key word READ or DATA, and a blank following the key words; and it is then followed by variable name(s) appropriate to the type of data to be read. The variable must be separated by a comma.

The data list is a list of data values, one value for each variable in the READ statement, separated by commas.

The data value must match the type and the order of the variables in the READ statement. For example,

PROGRAM 4.7

```
10  READ A1,B1, A$
20  DATA 20,30, Smith
30  PRINT A1,B1,A$
40  END
```

OUTPUT 4.7

```
20          30          smith
```

PROGRAM 4.8

```
10  READ  A1,B1,C2,A$,B$
20  DATA  20,30,40.10, Smith, Job
30  PRINT  A1,B1,C2,A$,B$
40  END
```

OUTPUT 4.8

```
20          30          40.1          smith          Job
```

Note that when two or more data values occupy a line, they are separated by commas. Character strings may or may not be enclosed in quotation marks in DATA statements. However, if the character strings are to contain leading or trailing blanks, commas, and/or semicolons, they must be enclosed in quotation marks.

In the Program 4.7, the READ statement in line 10 will assign values in line 20 to the variables A1, B1, and A$. The READ statement informs the computer to create a memory storage named A1 and take the first value in the DATA statement and put it in the A1 storage memory (Keep in mind that if there was a variable named A1 previously created and had a value in it, the variable value will be replaced by the new information). The second variable B1 (memory storage) will store 30, and the third variable will store Smith.

It is not necessary to group the variables and data as shown above. That is, more than one DATA statement may be used to satisfy one READ statement and more than one READ statement may be satisfied from one DATA statement. For example, the following statements are also acceptable:

PROGRAM 4.9

```
10 READ A1,B1,C2
20 READ A$,B$
30 DATA 20,30,40.10,Smith,Job
40 PRINT A1,B1,C2,A$,B$
50 END
```

OUTPUT 4.9

```
20          30          40.1          smith          Job
```

PROGRAM 4.10

```
10 READ A1,B1,C2
20 READ A$,B$
30 DATA 20,30,40.10
40 DATA Smith,Job
50 PRINT A1,B1,C2,A$,B$
60 END
```

OUTPUT 4.10

```
20          30          40.1          smith          Job
```

The variables in the above READ statements match the numeric and string variables in the DATA statement. That is, A1,B1,C2 are numeric variables which match with 20, 30, 40.1 respectively, and string variables A$,B$ match "Smith", and "Job" respectively. To understand how the data is read, it is helpful to think of a data pointer that moves through the data list and points to the data value that is to be read next. In the statement number 10, the data pointer is pointed at the first item in the data list. For the READ statement 20, the data pointer will be at "Smith." Keep in mind that some systems will require the string constant to be enclosed in quotes. That is, the above data statements 20, 30, and 40 might have to be written as follows:

```
20 DATA 20,30, "Smith"
```

OR

```
20 DATA 20,30,40.10,"Smith","Job"
```

OR

```
30 DATA  20,30,40.10,"Smith","Job"
```

OR

```
40 DATA  "Smith", "Job"
```

The DATA statement could be placed anywhere in the program between the END and the declaration statements. However, it is customary to place all the DATA statements at the beginning of the program or at the end of the program just before the END statement. However, the READ statements, like INPUT, are located wherever the logic of the program indicates the need for data. The READ statement should occur somewhere before the LET statement in the program where the variables to be read by the READ statement are used. The BASIC interpreter or compiler simply takes all the data items and forms one combined data list, ordering the DATA statements from lowest line number to highest and then using the data from left to right. For example, the following three statements look different, but the data lists they produce are alike:

```
10 INPUT A$,A1,B$,B1
10 DATA "Job",66
20 DATA "Carl",55
     OR
10 DATA "Job",66,"Carl",55
```

Results Stored as:

Storage	
A$	Job
A1	66
B$	Carl
B1	55

To create a READ/DATA statement, one should follow these rules for writing the READ statement:

READ/DATA Statement RULES

1: Every variable whose value is obtained by a READ should be listed in a READ statement prior to its being used elsewhere in the program.

2: A READ statement in a program must also have at least one DATA statement with matching values to be assigned to the variables listed in the READ statement.

3: Only commas should be used to separate values and variables

4: No comma should be placed at the end of the READ or DATA statement. A comma following a variable means the existence of another variable for which data to be read.

The examples would indicate the variables in the READ and the DATA statements must match perfectly. However, the BASIC is looking only for variables in the READ statement. That is, there should be at least as many data items in the DATA statement as the number of variables in the READ statement. Therefore, having more data items than the number of variables in the READ statement does not cause any error. For example,

PROGRAM 4.12

```
10  READ A1,B1,C1,C2
20  DATA 30,10.10,35,40,60,70
30  PRINT A1,B1,C1,C2
30  END
```

In this example, there are four variables in the READ statement and six data items in the DATA statement. Obviously, there are two more data items than the number of variables in the READ statement. This would not cause error. The computer will stop reading the additional data after the data for variable C2 is read. The output in this program looks as follows:

OUTPUT 4.12

```
30              10.10           35              40
```

On the other hand, an error will occur if there are fewer items in the data statement that the number of variables in the READ statement. For example,

PROGRAM 4.13

```
10  READ A1,B1,C1,C2
20  DATA 30,10.10,35
30  PRINT A1,B1,C1,C2
30  END
```

When this program is run, the OUT OF DATA in 10 message will be printed. That is, there are no data items for variable C2. After reading data item 10.10 for variable C1, it moves to read C2. However, when it comes to read the data item in the DATA statement, there is no more data available

to be read. Thus, the OUT OF DATA message is printed. The output of this program would looks as follows:

```
            OUTPUT 4.13

OUT OF DATA AT LINE 20
```

A message indicates that the end of the data list has been reached when a READ statement attempted to read a data list that has been exhausted. The message points out the line number of the READ statement in error.

```
            PROGRAM 4.14

100 PRINT "Enter the customer's name";
110 INPUT N$
120 PRINT "Fnter quantity";
130 PRINT K1
140 READ P,I
150 DATA 221,5
160 T= P*K1
170 PRINT  N$
180 PRINT "Quantity ", K1, " Price per unit $", P
190 PRINT "Total Amount $ ", T
200 END
```

```
                    OUTPUT 4.14

Enter the customer's name ? SHAH
Enter Quantity? 50
SHAH
Quantity       50      Price per unit    $  221
Total Amount $  11050
```

In all these examples, each DATA statement has one matching READ statement. That is, each DATA statement is processed by a READ statement only once. However, if one wants the system to read the same data items later, one must use the RESTORE statement to restore the data.

RESTORE Statement

Generally each READ statement has its own associated DATA statement, as the READ statement reads its data values from its DATA statement. But in some cases, you may want to reuse the same data values by another READ statement. You can accomplish it by either retyping the data in another DATA statement to match the READ statement. However, a better approach is to use the RESTORE statement. The RESTORE statement allows the data in a given program to be reread as often as desired by other READ statements. The RESTORE statement consists simply of the keyword RESTORE. The general form of the RESTORE is:

Ln RESTORE

Most large computers (mainframes) use this form. The RESTORE statement causes the pointer to be moved back to the beginning of the data sequence holding area. This is done so

that the next READ statement executed will read the data from the beginning of the sequence once again. For example:

PROGRAM 4.15

```
10 READ A1,B1,C2
20 READ A$,B$
30 DATA 20,30,40.10,Smith,Job
40 PRINT A1,B1,C2,A$,B$
50 RESTORE
60 READ A1,B1,C2
70 READ A$,B$
80 PRINT A1,B1,C2,A$,B$
90 END
```

In the above program, the READ statements 10 and 20 will read data line number 30, and then the RESTORE will restart the data line 30 again and the READ statements 50 and 60 will read the same data again.

OUTPUT 4.15

```
20        30        40.10        Smith        Job
```

The general form of the RESTORE for microcomputer is:

Ln RESTORE line number of the DATA statement

This format allows you to select the DATA statement where the next READ statement will read its data values. That is, by specifying the DATA statement number, you can select the DATA statement where the next data will be read. For example,

PROGRAM 4.16

```
10 READ A1,B1,C2
20 READ A$,B$
30 DATA 20,30,40.10
40 DATA Smith,Job
50 PRINT A1,B1,C2,A$,B$
60 RESTORE 40
70 READ C$,D$
80 PRINT C$,D$
90 END
```

In the above program, the READ statements 10 and 20 read the data in DATA statement 30. The PRINT statement 50 prints the values. Line number 60 restores the DATA statement 40, and the READ statement 70 reads the two names. The output would look as follows:

OUTPUT 4.16

```
20      30      40.10      Smith      Job
Smith           Job
```

The RESTORE statement is generally used when it is necessary to perform several types of computations on the same data items.

The PRINT Statement

A program is written to solve a problem. However, the program is of no value unless it shows the results on the screen or on paper. To accomplish this, the PRINT statement should be used. The format of the PRINT statement is a follows:

```
Ln    PRINT    "message" and/or list of variables
```

The PRINT statement is preceded by a statement number, a blank, and a blank following the keyword PRINT; and it is then followed by variable name(s) appropriate to the type of information already stored in the memory of the computer under the variables names to be printed. The variable names are separated by either a comma or semicolon.

For example,

PROGRAM 4.17

```
10 I  = 10
20 K  = 30
30 K1 = 40
40 L1 = 60.10
50 D2 = 44.20
60 PRINT  I
70 PRINT  I,K
80 PRINT  K1,L1,D2, E3
90 END
```

The PRINT statement causes the computer to bring the result out from the memory of the computer and displays the value(s) of the variable(s) on the screen. That is, the PRINT statement is used to display memory storage values, and to display heading or message. The PRINT statement can also be used to control the format of how the output should look, and the spacing of the output.

In the above example, statement 60 will display the value of I on the first line; statement 70 will display the values of I and K on the second line; statement 80 will display the values of K1, L1, D2, and E3 on the third line. The output of the above program will look as follows:

```
                    OUTPUT 4.17

     Column Numbers To Indicate where Values are Printed
              1         2         3         4         5         6
123456789012345678901234567890123456789012345678901234567890


 10
 10            30
 40            60.1          44.2          0
 10            30            40          -60.1
```

Keep in mind that values of the variables must have been assigned; that is, variables I, K, K1, L1, D2, E3 must been defined previously either through a READ or an INPUT or LET statement, e.g. lines 10 through 50. In the above program, the value of E3 is undefined. Therefore, the computer has assigned it a value of zero (0).

Commas or semicolons are placed between variables. A comma separator allows the output to be produced in a tabular form determined by the print zones of the system. That is, the PRINT statement prints one print line unless the number of variables exceeds the number of the print zones or

the PRINT statement ends with a semicolon. The comma displays the output at the next available print zone. On the other hand, the semicolon displays the output at the next available position and not at the next print zone. The semicolon computer instructs the computer to keep the print in the next available print position and not to move to the next print zone. For example:

```
PROGRAM  4.18

 10 I  = 10
 20 K  = 30
 30 K1 = 40
 40 L1 = 60.10
 50 D2 = 44.20
 60 PRINT  I
 70 PRINT  I;K
 80 PRINT  K1;L1;D2;E3
 90 END
```

The output of the above program will as follows:

```
                         OUTPUT 4.18

   Column Number To Indicate Where the Values are Printed

         1         2         3         4         5         6
123456789012345678901234567890123456789012345678901234567890
 10
 10  30
 40  60.1  44.2  0
```

In this example, the values are printed close to each other instead of within the 14 positions available for each print zone. The reason for the difference is that in the above PROGRAM 4.16, commas were used while PROGRAM 4.17 used semicolon.

The PRINT statement in BASIC displays the values of the variables into print zones. Most BASICs establish a print area of a specific number of characters and divide them into print zones. The size of print zone varies among systems. However, most computer systems use five print zones per line. Each print line has 70 positions divided into 14 positions per zone. Each print zone specifies an area for a variable to be printed or displayed.

The following rules are applied when data is printed by a PRINT statement with a comma as a separator.

PRINT ZONE RULE WITH COMMA AS A SEPARATOR

Each PRINT statement displays one line, unless

1. The PRINT statement has more than five variables. Line 130 in PROGRAM 4.19 shows this exception. In such cases, the first five variables are shown in the first print line, the next five in the next line and so on.

2. The PRINT statement has a comma as a last character. For example, line 140 in PROGRAM 4.19 ends with a comma. The variable in line 150 will be printed on the same line but at the next available print zone.

The zones are numbered consecutively, starting with position one, as follows:

```
1 <------>14 15<------>28 29<------>42 43<------>56 57<------>70
   print        print        print        print        print
   zone 1       zone 2       zone 3       zone 4       zone 5
```

PROGRAM 4.19

```
  1 PRINT "         1         2         3         4         5"
  5 PRINT "12345678901234567890123456789012345678901234567890"
 10 LET   I = 30
 20 LET   K = -50
 30 LET   K1 = 40.00
 40 LET   L1 = -6000.
 50 LET   D2 = -99999
 60 LET   E3 = 4.99999980
 70 LET   M4 = 667
 80 PRINT I
 90 PRINT I,K
100  PRINT I,K,K1
110  PRINT I,K,K1,L1
120  PRINT I,K,K1,L1,D2
130  PRINT I,K1,L1,D2,E3,M4
140  PRINT I,K1,
150  PRINT L1,D2,E3
160  END
```

The form of output to be displayed depends on the values to be displayed. That is, an integer (whole number) is displayed without a decimal. A negative integer number is displayed with the negative sign before the number; a positive integer number is displayed with a blank preceding it. For example, the output of the above program would be displayed as follows:

OUTPUT 4.19

```
Print   Zone 1 Print  Zone 2  Print Zone 3  Print Zone 4 Print Zone 5
---------------------------------------------------------------------
123456789012345678901234567890123456789012345678901234567890123456789
 30
 30            -50
 30            -50            40
 30            -50            40            -6000
 30            -50            40            -6000           -99999
 30             40           -6000          -99999           5
```

However, it is advisable to have a message with the printed value. The PRINT statement could be used to print a message or a message and the variables' values. For example to rewrite (3.8):

PROGRAM 4.20

```
 10 LET   I = 30
 20 LET   K = 50
 30 LET   K1 = 40
 40 LET   L1 = 60
 50 LET   D2 = 70
 60 LET   E3 = 80
 70 PRINT " VALUE OF I ", I
 80 PRINT
 90 PRINT " VALUE OF I ", I, " VALUE OF K ", K
100 PRINT " VALUE OF K1 ", K1, " VALUE OF L1 ", L1
110 PRINT " VALUE OF D2 ", D2, " VALUE OF E3 ", E3
120 END
```

These statements will print the results as follow:

OUTPUT 4.20

```
VALUE OF I     30

VALUE OF I     30             VALUE OF K    50
VALUE OF K1    40             VALUE OF L1   60
VALUE OF D2    70             VALUE OF E3   80
```

The blank line can be printed by using the PRINT without message or variable(s) list. In the above program, line 90 will print a blank line as shown in the output. The PRINT statement also allows calculation to be made within the PRINT statement. For example,

PROGRAM 4.21

```
10 LET L = 10
20 PRINT  6/7, 10*2, L
30 END
```

This PRINT statement has two equations and a variable. That is, the first equation divides 6 by 7, the second equation multiplies 10 by 2, and the variable L is printed from the memory. These statements will give the following output:

OUTPUT 4.21

```
.8571429        20              10
```

In addition, the PRINT statement could be used as a calculator. If one needs quick calculations without having to create and run a program, the PRINT statement could be used for that purpose. Any time a quick calculation is needed, use the PRINT statement without a line number. That is,

```
PRINT 1000/5, 6*6, (5+4)^2
```

This statement will print the following results immediately after pressing the RETURN key:

```
        200         36          81
```

This type of PRINT statement could be used anytime, especially during debugging a program. In order to print the values of the variables executed during the execution of the program, one just types the PRINT statement with the list of variables.

The TAB Function in PRINT Statement

The PRINT statement controls the vertical output of the program. With the proper use of commas and semicolons within the PRINT statement, the output could be created with proper spacing. However, the same result could be achieved with the TAB function which helps control the horizontal spacing of the output. The TAB function embedded in the PRINT statement specifies the exact print position of the output on a given line. In effect, the TAB function moves the printer or the cursor to a specified position. That is, the TAB function operates a tabulator key on a typewriter.

The general form of the TAB function is:

```
Ln    PRINT    TAB(numeric expression), variable list
```

The PRINT statement is preceded by a statement number, blank, and a blank following the keyword PRINT; and it is then followed the keyword TAB with a numeric value or an expression within the parenthesis, and by variable name(s) appropriate to the type of information already stored in the

memory of the computer under the variable names to be printed. The variable names are separated by either a comma or semicolon. However, use semicolon between variables. If comma is used, it would print the next variable in the next print zone.

The numeric expression may be an integer value, an expression, or a variable. The value of the expression determines the print position relative to the starting position of the line.

The print positions are numbers beginning with 1 on the left. For example,

PROGRAM 4.22

```
10  L = 10
20 PRINT TAB(10),"John Doe",TAB(30),"Preston Street",TAB(55),L
30  END
```

will display output as

OUTPUT 4.22

```
                    Column number
         1         2         3         4         5         6
123456789012345678901234567890123456789012345678901234567890
         John Doe                    Preston Street
```

The message of statement 10, "John Doe," starts at column number 10 and continues until the message is displayed completely. That is, the last letter, the "e," is printed at column 17, a total of eight characters. Then the message "Preston Street" is printed starting at position 30. However, where TAB function is used with numeric value, the first digit position on the left is left blank for the sign preceding the number. It is very important that the TAB function is used with caution. That is, count the spacing for each variable carefully before creating the program.

If spaces are miscalculated, the output will be incorrect. For example, to rewrite PROGRAM 4.22:

PROGRAM 4.23

```
10  L = 10
20 PRINT TAB(10),"John Doe",TAB(30),"Preston Street",TAB(55),L
30  END
```

will result in the following output:

OUTPUT 4.23

```
                     Column number
         1         2         3         4         5         6
123456789012345678901234567890123456789012345678901234567890
         John DUSA           Preston Street
                                                        10

                        or

         John Doe
               USA           Preston Street
                                                        10
```

The letters "oe" of "Doe" of the name John Doe are truncated and overwritten by the word USA. In the print statement, without a proper count of spacing, not enough spaces (at least nine spaces) were provided for John Doe. As a result, when the next TAB function was used, the word USA was printed at location sixteen (as specified by the TAB(16) -- thus overwriting the last two letters of the word DOE. However, in some cases instead of overwriting the previous value, the computer will print the word USA in the 16th column of the next line output. Further, if there are not enough spaces left on the line to fit the value of a variable, the value will be placed on the next output line.

A variable can be used to tell the computer where to print. For example,

PROGRAM 4.24

```
10  M = 10
20  N = 10
30  PRINT TAB (N);"SHAH";TAB(N*2);"COMPANY"
40  PRINT TAB (5);"______________________________"
50  PRINT TAB (15);"   SALARY TABLE"
60  PRINT TAB (5);"NAME";TAB(M);"SALARY"
70  PRINT TAB (5);"______________________________"
80  END
```

In the above program, variables and an expression are used in the TAB function. In line 30, TAB(N) indicates to print SHAH starting in column 10, and TAB(N*2) indicates to print COMPANY starting in column 20 defined by N times 2 or (10 x 2 = 20). This is an acceptable way to use the TAB function.

PRINT USING STATEMENTS

The PRINT USING statement provides for better control of the output than any of the methods discussed so far. The PRINT USING statement uses a format character in the form of symbols to specify format. The PRINT USING provides for:

a) exact image of a line of output;
b) alignment of decimal and the number of digits to be displayed when printing columnar tables with numeric;
c) specification of commas within a number, the sign of the number, and the dollar ($) sign;

The general form of the PRINT USING is:

Ln PRINT USING format number; variable(s) list

or

Ln PRINT USING string expression; variable(s) list

The PRINT USING statement consists of the keywords PRINT and USING followed by the format enclosed in quotation marks, followed by a semicolon; and the list of variables separated by a comma.

These are the most common forms of the PRINT USING statements. Most of the microcomputers and the IBM systems use the second form. The other mainframe systems use the first form. One should check the system's manual for an appropriate form. However, the second form will be discussed here. The format to be used with the PRINT statement must use certain standard symbols.

They are:

Symbol	USE
#	This symbol is used to specify a numeric field. One symbol is needed for each digit. For example: ### allows for three digits ####.## allows for six-digit number, four digits to the left of the decimal and two to the right of the decimal The period symbol is used for placing a decimal within a number. If a number is a floating (real) number, you must provide for a decimal.
+ or -	The plus or minus sign indicates the status of a number. For example: -####.## a number will be printed with a negative sign +### a number will be printed with a positive sign
$	The dollar sign as the first character displays the number with a dollar sign. For example, $####.##
\\	The backslash separated by an appropriate number of blank spaces determines the number of characters to be printed for alphabetic or alphanumeric variables. The two backslashes account for two character spaces.

For example:

```
column numbers
123456789
\      \ this format provides for 8 characters, i.e.,
        six blank spaces between the two backslash.
```

If the number of characters in the variable memory is less than the size of the format, the left most spaces are filled only. If the number of characters in the variable memory is more than the size of the format, the right most characters are cruncated.

A comma is used to place a comma within a number.

For example

###,###,###.##

For example, a program containing all these formats could be written as follows:

PROGRAM 4.25

```
10  PRINT  "\                 \"; "INTEREST TABLE"
20  PRINT  "#######.##    ####.##    ######.##"; P,A,I
30  PRINT  "#,###,###.##  #,###.##  ###,###.##";P,A,I
40  PRINT  "\                 \"; "Interest Amount is",A
```

```
OUTPUT 4.25

INTEREST TABLE
  90000.00    5400.00              0.06
  90,000.00  5,400.00              0.06
Interest Amount     5400
```

Example of format:

Field Format	Variable or Constant	Output	Discussion
#####	56789	56789	five-digit constant printed as is
#######	56789	bb56789	five-digit number in seven spaces
###	56789	%56789	five-digit number to be printed in three spaces, not enough spaces for the number. This is indicated by % (percent) sign in front of the number.
###.##	567.89	567.89	five-digit decimal number
###.##	567	567.00	trailing zeros added
###.##	67.686	67.69	decimal fractional rounded
##,###.##	56789.9	56,789.90	number with a comma
##,###.##	789.9	bbb789.90	leading spaces are left blank
$#,###.##	56789.9	$56,789.90	number with a $ sign
$#,###.##	6789.9	$ 6,789.90	number with a $ sign, a space between the $ sign and the first digit.
$$,###.##	6789.9	$6,789.9	format with two $ signs. Number with a $ sign, and the $ sign is placed next to the first digit.
\ \	XYZ	XYZ	size fits the number of characters
\ \	XYZ	XYZbbbb	three characters in seven spaces
\ \	UNITED STATES	UNITEDb	thirteen characters in seven spaces Last word STATES truncated.
\ \	PAKISTAN	PAKISTAN	eight characters in eight spaces
\ \	PAKISTAN	PAKIS	only the first five characters are printed

Make sure to provide enough symbols in the format statement to accommodate the largest number or string that is to be printed in that field. If the number to be printed is less than the spaces provided, the number will be printed in the rightmost position. If the number to be printed is greater than the spaces provided, the number will be truncated and error message, or % (percent) sign will be printed at the beginning of the number. In the case of characters, they are printed at the leftmost position, if more spaces are provided than the number of characters to be printed. However, the rightmost characters are truncated if fewer spaces are provided that the number of characters to be printed.

LPRINT

In programs where output has to be printed on the printer instead of being displayed, it is advisable to use the LPRINT or LPRINT USING statement rather than PRINT. The general form is the same as PRINT. However, keep in mind that there will be no output displayed on the screen.

QUESTIONS

1. What are some of the ways a BASIC program can be programmed to check for errors in data?

2. If the computer prints "Not Enough Data," what could be the possible reasons for getting this message? How would you solve this problem?

3. What does the "Out of Data" message mean?

4. What does the "Redo" message mean, and what causes this message?

5. What are the differences between logical errors and syntax errors?

6. Describe the three ways of inputting data into a Basic program?

7. Correct errors, if any, in the following statements.

```
a)  INPUT N, M/N
b)  INPUT 2N, 2, N
c)  INPUT N, M, C
d)  INPUT N*N, M1, C23, XYX, N+2
e)  INPUT N;M;K
f)  INPUT N,M,K,
g)  INPUT N$,M, K1
```

8. Write a program segment that asks for a date as a string in the form of dd/mm/yy, and then prints it.

9. Discuss situation(s) where the INPUT statement will be more appropriate than the READ/DATA statement.

10. Write a program segment where inputting data is indicated by a message that tells you what to enter.

11. What is interactive programming? Discuss its advantages.

12. Write a program segment which reads names, ID numbers, and pay rates of employees and prints them.

13. Correct errors, if any, in the following statements.

a)
```
READ   N,M,I
DATA   10, 20
```

b)
```
READ   N,M
DATA   10, 20, 30
```

c)
```
READ   N$, P
DATA   Syed, 10
```

d)
```
READ   N$, P
DATA   10, Syed
```

e)
```
READ   N,M,K
```

14. What is the RESTORE statement used for?

15. Correct errors, if any, in the following statements. Assume that the variables used in the PRINT statement are already defined (known to the computer).

```
a)  PRINT  N, M, I
b)  PRINT  "KARACHI, M, 10/3
c)  PRINT  N$, "SMALL", 5
d)  PRINT N$;;"PAKISTAN";10;5,6
e)  PRINT
```

16. Explain how the print zones work.

17. What is the purpose of the TAB function? Write a program segment which shows the use of the TAB function.

18. Describe a situation where you will use the PRINT USING statement.

19. What does the LPRINT statement do?

20. Suppose your teacher gives five examinations worth 100 points each, assigns 20 homework problems worth 10 points each, and assigns 100 points to class attendance during the year. Write a program that reads and prints the names of the students, their examination grades, homework points, class attendance points, and calculates and prints the average grade for each student.

21. Write a program that calculates and prints interest for any loan amount, the duration of loan, and the interest rate.

22. Write a program that calculates the Body Mass Index. The Body Mass Index is used as a standard for determining body weight for men or women in a country. For example, the Body Mass Index is 27.8 for men and 27.3 for women in the United States. The equation for calculating the Body Mass index is :

$$\text{Weight (in kilograms)} - \text{Height}^2 \text{ (in meters)} = \text{index}$$

For example, a person weighing 200 lbs. (90.91 kilograms), and is 6 feet (1.83 meters) tall would have 27.20 as a Body Mass Index. The calculation is shown as:

$$90.91 - (1.83)^2 = 90.91 - 3.342 = 27.20$$

23. Write a program that prints address labels.

24. In 1988, the population of Pakistan was estimated to be 100,000,00, and was growing at the rate of 3.25 percent a year. Write a program that will project the population into the year 2010. For example,

$$P = P * (1 + .0325)^{22}$$

25. Write a program that prints the following letter as is.

```
             a
            aaa
           aa aa
          aa   aa
         aa     aa
        aa       aa
       aaaaaaaaaaaaa
      aa           aa
     aa             aa
    aa               aa
   aa                 aa
  aa                   aa
 aa                     aa
aa                       aa
```

CHAPTER 5

STRUCTURED PROGRAMMING AND THE CONDITIONAL APPROACH

PROGRAM REPETITION

So far every program shown could use one set of data or input in each run. That is, after the command RUN is typed, the program will be executed from the beginning to the end in sequential order and will stop execution. However, there are situations where you may want to input more data or let the same set of statements read and execute more data, or change the sequence of the execution in the program. For example, the following program will run only once:

PROGRAM 5.1

```
10 INPUT "Type the amount of loan"; P
20 INPUT "Type the interest rate"; I
30 A  = P * I
40 PRINT "Amount "; A;"Loan "; P ; "Interest rate ";I
50 END
```

The statements discussed so far do not provide for such a possibility. Therefore, we need to learn a method by which we can make the following program repeat itself and/or change the sequence of the execution of the program. That is, it should ask for more data input without typing the command RUN.

To accomplish program repetition or change the flow of execution, there are many statements in BASIC, such as GOTO, IF.. THEN, IF.. THEN ... ELSE, ON..GOTO, and others.

GOTO Statement

The GOTO (or GO TO) is one of the BASIC statements which breaks unconditionally the sequence of execution of the program. The purpose of the GOTO statement is to break the normal sequential execution of the program and continue execution at the statement specified in the GOTO statement. This transfer of control to another point in the program is called **branching** or **control transfer.** The general form of this statement is:

```
Ln1  GOTO  Ln2
```

The (Ln1) is the line number of the GOTO statement; the keyword is GOTO, which can be written as one word or as two words, GO TO; and the line number (Ln2) follows after the word GOTO. The last line number in the statement refers to the line number of the statement to which the program will transfer control when the computer reaches the GOTO statement during the execution. For example,

```
10 GOTO 300
```

In this statement, the number 10 is the statement number of the GOTO statement in the program, and the statement number 300 in the statement is the statement number in the program where the computer will branch to. It is possible to branch out from either direction in the program -- that is, from the bottom of the program to the top of the program or from the top of the program to the bottom of the program. Make sure that there is a line number 300 in the program. If there is not such a line number in the program,

you will get an error of UNDEFINED LINE number. The GOTO statement can also be used for **looping**, which is the process of executing a group of statements repeatedly. For example,

PROGRAM 5.2

```
10 INPUT "Type the amount of loan"; P
20 INPUT "Type the interest rate"; I
30 A  = P * I
40 PRINT "Amount "; A;"Loan "; P ; "Interest rate ";I
50 GOTO  10
60 END
```

This program, a modified version of Program 5.1, can run repeatedly until interrupted (terminated) when the programmer presses a certain set of keys on the keyboard. This process of interruption of a program is called **abnormal termination**. In this program, the computer asks for the amount of a loan, and interest rate and prints the calculated interest amount, loan amount, and interest rate. At statement number 50, the program finds the GOTO statement and line number 10 within the GOTO statement. The execution is transferred back to line number 10. Since line number 10 is the first statement in the program, it starts the execution of the program again. Therefore, this program is in an infinite loop. That is, the program will continuously ask for data and will print the results unless interrupted by the programmer. Before running a program with an infinite loop, check your computer system's manual to learn how to stop the execution of such a program. Most systems use CRTL/BREAK keys or CTRL/T or other combinations of keys with CTRL. That is, hold down the CTRL key and then press the other specified key.

The output of the above program would look as follows, including the error message due to abnormal termination:

OUTPUT 5.2

```
Type the amount of loan ? 1000
Type the interest rate? .10
Amount  100 Loan 1000  Interest rate .1
Type the amount of loan ? 2000
Type the interest rate? .05
Amount  100 Loan 2000  Interest rate .05
Type the amount of loan? (CTRL/BREAK) keys were pressed
Break in 20  Note: This statement was the result
                   of an abnormal break.
```

The above output was the result of running Program 5.2 and inputting two types of input information. After the third input request, the CTRL/BREAK keys were pressed to terminate the program. This abnormal termination resulted in the error message " **BREAK IN 20.**"

However, the effect would have been different if the program had been data in it. For example, Program 5.3 is a modified version of Program 5.2, except in this program the data is read rather than inputted. The computer will read four sets of numbers and then will run out of data.

PROGRAM 5.3

```
10 READ P,I
20 A = P * I
30 PRINT "Amount "; A;"Loan "; P ; "Interest rate ";I
50 GOTO  10
60 DATA 10000,.10,9000,.09,5000,.06,12000,.15
70 END
```

In the above program, the READ statement reads two numbers each time it is executed. However, there are a total of eight numbers to be read. When it reads the last numbers (12000, .15) and comes back to read additional data, there are no more numbers to read. This results in the message "Out of DATA in 20." The output of the program is shown below.

```
                    OUTPUT   5.3

Amount     1000 Loan    10000 Interest rate   .1
Amount     810 Loan    9000 Interest rate   .09
Amount     300 Loan    5000 Interest rate   .06
Amount     1800 Loan    12000 Interest rate  .15
Out of DATA in 20  Note :  This message is the
                              result of the READ
                              statement not finding
                              additional data to read
```

The GOTO statement makes unconditional transfer or creates an infinite loop in the program. These are undesirable programming techniques. A better method is to control when or under what conditions a transfer should be made or when a loop should end.

Structured Programming

It may be appropriate to discuss structured programming at this time. Most novice programmers consider programming as creating a set of statements to be followed by the computer during the execution of the program. The steps required to write a program were discussed previously. However, a good programmer would follow a **structured programming** procedure. Structured programming is a systematic process which requires writing programs that are well structured. The objectives of a well structured program are: increased clarity, reduced testing time, easier maintenance, and more accurate results. In short, structured programming is easier to follow.

These objectives could be achieved by using a **top-down modular programming** approach. This means breaking down the program into modules or steps which make up the program. Further, structured programming requires the use of one of the following control structure techniques for a well structured program:

1. **Sequence:** This requires putting all related statements in the order of their execution. That is, there is no statement transferring execution.

2. **IF..THEN..ELSE:** This provides alternative actions for execution. In this control statement, the computer tests the IF..THEN..ELSE condition for "true" or "false." If the condition is true, the execution is transferred to the statement number after the keyword THEN, otherwise to the statement number after the ELSE.

3. **FOR..NEXT:** This provides for a specified number of repeated executions of a set of statements. DO..WHILE and DO..UNTIL are other types of statements which do repeated executions.

However, structured programming could be written with little or no use of the GOTO statement.

One of the BASIC control statements listed above is the IF..THEN statement.

IF..THEN Statement

If at any time you want the computer to alter the normal sequential order of execution based on condition, you use the IF.. THEN statement. The general form of the statement is:

```
                                         Ln2
Ln1  IF  relational expression THEN      or
                                         statement
```

The statement consists of the statement number (Ln1) and the keyword IF followed by a relational expression which is followed by the keyword THEN, and either a statement number to which the execution will be transferred if the condition is true or a BASIC statement. An example of the IF..THEN with a statement number is:

```
20  IF  A>15 THEN  50
```

The statement consists of statement number 20, the keyword IF, a relational expression (A>15), the keyword THEN, and a statement number 50. In this statement, if the value of A (undefined for now) is greater than 15 (a true condition), the execution will transfer to statement number 50 in the program. Otherwise, if the condition is false (if A is not greater than 15), then the statement following this statement is executed.

An example of the IF..THEN statement with a statement or message is:

```
10 IF  A>15 THEN  PRINT "A IS GREATER THAN 15"
```

or

```
10 IF A>15 THEN B = 60
```

In this example, instead of transferring control to another statement in the program, it executes the statement after the THEN if the A is greater than 15. Otherwise, the statement following this statement is executed.

The IF..THEN statement is not limited to just numeric comparison. It could contain comparison of constants, algebraic expressions, or numeric variables. Also, string constants and variables can be compared for their ascending or descending order.

Table 5.1 discusses several examples:

TABLE 5.1

EXAMPLES of the IF..THEN Statement

Statements

```
10    A = 10
20 IF A = 10  THEN 50
```

Purpose : comparing a variable A against constant number 10.

Result : case of true condition, the program will transfer to statement number 50.

```
10   A = 10
20   B = 20
30   IF A>B THEN PRINT " A IS GREATER THAN B"
```

Purpose : comparing variable A against variable B.

Result : case of false condition, the program will not print "A IS GREATER THAN B".

```
10  IF 10 > 20 THEN PRINT " 10 > 20"
```

Purpose : comparing two constants.

Result : case of false condition, the message " 10 > 20" will not be printed.

```
10  A = 2
20  B = 3
30  IF 40 < A*B THEN  PRINT  "40 IS GREATER THAN A TIMES B"
```

Purpose : comparing constant with expression.

Result : case of true condition, the message will be printed.

(continued on next page)

```
10   A = "SHAH"
20   IF A<>"SHAH" THEN  60
```

Purpose : string constant against string variable.

Result : case of false condition, the statement following this statement will be executed.

```
10   IF   "SYED">"SHAH" THEN PRINT "SYED IS GREATER THAN SHAH"
```

Purpose : string constant against string constant.

Result : case of true condition, the message will be printed.

In the last example in Table 5.1, the BASIC determines which characters are "less" than other characters using the ASCII or EBCDIC codes. In ASCII, numbers are less than letters, and some special symbols are less than numbers. Among letters, A is less than B, B is less than C, and so on. Therefore, SYED is greater than SHAH.

The IF..THEN statement can also be used to test for the end of a loop or end-of-file (EOF) condition. In the case of the READ/DATA statements, the IF..THEN statement could be used to look for a last data item that is disguishable from the rest of the data in the DATA statement. This is called a **trailer record or card** (data item). It is also called a **sentinel record**. After each READ statement, the IF..THEN statement tests for this unique value. When the value is detected, the execution is stopped or transferred to another section of the program. For example,

PROGRAM 5.4

```
10 READ P,I
20 IF P = 0 THEN 70
20 A = P * I
30 PRINT "Amount "; A;"Loan "; P ; "Interest rate ";I
50 GOTO  10
60 DATA 10000,.10,9000,.09,5000,.06,12000,.15,0,0
70 END                                  [sentinel values]
```

In the above program, the last two values (0,0), sentinel values, in the DATA statement are unique values. That is, when the computer reads these two values, one for each variable P and I, the condition in statement number 20 is satisfied and the control is transferred to statement number 70. Since statement number 70 is the END statement, the execution of the program is terminated. The output of the program is as follows:

OUTPUT 5.4

```
Amount    1000 Loan   10000 Interest rate    .1
Amount    810 Loan   9000 Interest rate    .09
Amount    300 Loan   5000 Interest rate    .06
Amount    1800 Loan   12000 Interest rate   .15
```

The above output shows no abnormal termination message. The program ended under normal conditions.

In the case of the INPUT statement, when the programmers do not want to enter additional data, they just enter the unique (sentinel) value and the execution is stopped or transferred to another section of the program. For example,

PROGRAM 5.5

```
 10 PRINT "Type the loan amount. Type 0 (zero) to stop";
 20 INPUT  P
 30 IF  P = 0 THEN 100
 50 PRINT "Type the interest rate.";
 60 INPUT I
 70 A = P * I
 80 PRINT "Amount "; A;"Loan "; P ; "Interest rate ";I
 90 GOTO  10
100 END
```

Anytime you want to stop the above program, just type zero (0) as a value for the loan amount. The output of the above program is shown below:

OUTPUT 5.5

```
Type the amount of loan. Type zero (0) to stop ? 1000
Type the interest rate? .10
Amount   100 Loan  1000 Interest rate  .1
Type the amount of loan. Type zero (0) to stop ? 2000
Type the interest rate? .05
Amount   100 Loan  2000 Interest rate  .05
Type the amount of loan. Type zero (0) to stop ? 0
```

The above output shows that at the last input statement, the value of zero (0) was typed which terminated the program.

In some cases it might be appropriate to let program count how many times it has run and stop when the specified number is reached. This process is called **counting loop** and requires initializing a variable as a **counter** before it enters the loop. Counters are usually initialized to a value of zero, but, this is not required. The initial value could be anything. In the loop, the value of the counter is modified (increased or decreased) every time the loop is executed. The counter is tested in the IF..THEN statement to determine whether the loop has exceeded the specified number.

As an example, suppose we invested Rs. 1,000 in a bank for 5 years at 10 percent interest rate compounded annually. How much will our money be worth each year? As you can see, this program requires a loop to calculate the interest amount for each year for five years. Obviously, it also requires an equation for compounding interest, i.e.

Amount = Principal * (1 + interest rate) ** year(s).

In addition, we will keep a counter (running total) of the number of years in order to stop the calculation of the interest amount when we reach five years. The program would look as follows:

PROGRAM 5.6

```
10  PRINT "         INTEREST        "
20  PRINT "YEAR      AMOUNT      BALANCE"
30  PRINT "---------------------------"
40  Y =  1
```

(continued on next page)

PROGRAM 5.6 (continued)

```
 50  READ P,I
 60  A = P
 70  A = A * (1 + I) ** Y
 80  B = A - P
 90  PRINT USING;"###     #####.##      ######.##";Y;B;A
100  Y = Y + 1
110  P = A
120  IF Y <= 5 THEN 70
130  DATA 1000,.10
140  END
```

In this example, the counter, variable Y, is initialized in line number 30. The initial value assigned to Y is 1. In line number 60, variable A is initialized to the amount of the principal. In line 70, the interest is compounded. In line 80, the interest amount is calculated by subtracting the balance from the compounded amount. In line 100, the Y counter is updated by adding one year to the counter. This counter keeps a record of how many years the program has run. Line 110 tests the counter. If the counter exceeds five (5) years, the program stops. That is, the program does not go back to 70 but executes statement 120, the END statement.

The output of the program is:

OUTPUT 5.6

YEAR	INTEREST AMOUNT	BALANCE
1	100.00	1100.00
2	231.00	1331.00
3	440.00	1771.56
4	822.18	2593.74
5	1583.51	4177.25

However, counters have many uses other than just keeping a count of how many times the program has been executed. The counter can also be used to keep a running total of anything. For example, it could be used to total the salaries of the employees in the organization, to add up the grades of students in a course, and to find the average grade in the course. One can find many other uses for counters.

For example, to find an average grade of ten students in a course, one has to use counters. A program that requires inputting ten scores and then calculates the average grade for the class is as follows:

PROGRAM 5.7

```
 10  T = 0
 20  T1 = 0
 30  PRINT " TYPE THE GRADE OF A STUDENT";
 40  INPUT G
 50  T = T + 1
 60  T1 = T1 + G
 70  IF T < 10 THEN 30
 80  A1 = T1 / T
 90  PRINT "THE AVERAGE GRADE OF "; T ;"  STUDENTS IS "; A1
100  END
```

In this program, we need two counters: one for keeping a count of the number of students, and another to sum the grades. These two totals are needed because the average is calculated as:

Average = sum of all grades / total number of students

As mentioned previously, to create a counter, you initialize the counter first. Lines 10 and 20 are for initializing variables T and T1. After initialization, line 50 is keeping a count of the number of students, and line 60 is adding up the grades as they inputted. When the value of variable T exceeds 10, the next statement, line 80, is executed. Line 80 calculates the average.

That is, it divides T1, the total of all grades, by T, by the number of students. The output of the program is shown below:

OUTPUT 5.7

```
TYPE THE GRADE OF A STUDENT ? 66
TYPE THE GRADE OF A STUDENT ? 87
TYPE THE GRADE OF A STUDENT ? 85
TYPE THE GRADE OF A STUDENT ? 75
TYPE THE GRADE OF A STUDENT ? 95
TYPE THE GRADE OF A STUDENT ? 100
TYPE THE GRADE OF A STUDENT ? 88
TYPE THE GRADE OF A STUDENT ? 57
TYPE THE GRADE OF A STUDENT ? 46
TYPE THE GRADE OF A STUDENT ? 50
THE AVERAGE GRADE OF   10   STUDENTS IS   74.9
```

The IF..THEN..ELSE Statement

Some versions of the BASIC will permit the IF..THEN..ELSE statement. The general form of the statement is:

```
                                        Ln2                 Ln3
Ln1  IF  relational expression THEN     or         ELSE     or
                                     statement          statement
```

The statement consists of the statement number (Ln1), the keyword IF followed by a relational expression, the keyword THEN, a statement number (Ln2) where a transfer will be made, or a BASIC statement. Then the keyword ELSE follows and either a statement number (Ln3) or a BASIC statement following the keyword ELSE.

In the case of the IF..THEN..ELSE statement, the relational expression is evaluated. If the condition is true, the statement number or the BASIC statement after the word THEN is executed. If the condition is false, the statement number or the BASIC statement after the keyword ELSE is executed. The important difference between IF..THEN and IF..THEN..ELSE is that in the case of IF..THEN the statement following the IF..THEN statement is executed when the condition is true. On the other hand, if the condition is false, the control passes to the statement number or the BASIC statement after the keyword ELSE. For example:

```
10  IF  10 > 20  THEN   A = 50  ELSE  B = 50
```

In this statement, the values 10 and 20 are compared. Since the condition requires that 10 be greater than 20, a false condition, the statement that follows the word ELSE will be executed. That is, the variable B will be assigned the value of 50. However, in this example, after the value is assigned to B, the control is not transferred to another statement, but the statement following this statement will executed. On the other hand, in the following example, the control will be transferred to another statement in the program:

```
10  IF  20 > 10 THEN  50  ELSE  90
```

If this statement, the condition is true, the control will be transferred to statement number 50, and the statement number 90 after the word ELSE will not be checked. However, as shown in Table 2.1, there are many variations of

the IF..THEN..ELSE statement. That is, the relational expression could be of many varieties, and the key word THEN could be followed by an expression and the key word ELSE could be followed by the statement number or vice versa. For example,

```
10  IF  A = B  THEN  C = 9 * 2  ELSE  90
```

In this statement, the variables A and B are compared. If A is equal to B, the statement C = 9 * 2 will be executed; otherwise, the control will be transferred to statement number 90, the statement number after the word ELSE.

To summarize all the BASIC control statements, let us write a program which will use all the statements learned so far.

Suppose you were to write a payroll program for hourly workers. Suppose, also, that your workers were paid double for overtime. That is, anyone working more than 48 hours a week will be paid overtime for the extra hours. To calculate a gross pay based on the hourly rate and the number of hours worked, we can write the following program:

PROGRAM 5.8

```
 10 PRINT"TYPE THE NAME OF THE EMPLOYEE. TYPE N TO STOP";
 20 INPUT N$
 30 IF  N$ ="N" THEN 130
 40 PRINT "TYPE THE NUMBER OF HOURS WORKED PER WEEK";
 50 INPUT H1
 60 PRINT "TYPE THE RATE PER HOUR";
 70 INPUT R1
 80 IF R1>48 THEN G=(48*R1)+(H1-48)*(R1*2) ELSE G=H1* R1
 90 PRINT "NAME                        "; N$
100 PRINT "TOTAL HOURS WORKED          "; H1
110 PRINT " TOTAL WAGES             Rs."; G
120 GO TO 10
130 END
```

In this program, line 30 checks for a condition to stop the execution of the program when the programmer types the capital letter N. Line 80 calculates overtime, if an employee has worked more than 48 hours, by using the equation

```
G = 48 * R1 +  (H1 - 48) * (R1*2)
```

after the word THEN. In this equation, a worker with overtime will be paid for the first 48 hours at his regular rate (48 * R1) and at the overtime rate for all the hours over 48 hours. However, if the worker has worked for 48 hours or less, then the equation

```
G = H1 * R1
```

after the word ELSE is used.

The output of the above program is as follows:

OUTPUT 5.8

```
TYPE THE NAME OF THE EMPLOYEE. TYPE N TO STOP ? Shah
TYPE THE NUMBER OF HOURS WORKED PER WEEK ? 60
TYPE THE RATE PER HOUR ? 5
 NAME                          Shah
 TOTAL HOURS WORKED             60
 TOTAL WAGES                Rs. 360

TYPE THE NAME OF THE EMPLOYEE. TYPE N TO STOP ? Ali Khan
TYPE THE NUMBER OF HOURS WORKED PER WEEK ? 50
TYPE THE RATE PER HOUR ? 6
 NAME                          Ali Khan
 TOTAL HOURS WORKED             50
 TOTAL WAGES                Rs. 300
```

(continued on next page)

```
                OUTPUT 5.8  (continued)

TYPE THE NAME OF THE EMPLOYEE. TYPE N TO STOP ? Ajeeb
TYPE THE NUMBER OF HOURS WORKED PER WEEK ? 40
TYPE THE RATE PER HOUR ? 5
 NAME                         Ajeeb
 TOTAL HOURS WORKED            40
 TOTAL WAGES               Rs. 200


TYPE THE NAME OF THE EMPLOYEE. TYPE N TO STOP ? N
```

In the above output, Shah worked 60 hours this week at an hourly rate of Rs. 5. He will be paid for the first 48 hours at the rate of Rs.5 per hour, a total of Rs. 240, and at the rate of Rs. 10 per hour for the additional 12 hours; a total of Rs. 120. Therefore, Shah's total wages for this week will be Rs. 360. However, Ajeeb will be paid a total of Rs. 200 for 40 hours at the rate of Rs. 5 per hour.

The last line in the output shows that the programmer stopped the program by typing the capital letter N instead of a name. This value, N, made the condition true in statement 30, and the program transferred the control to statement 130. Line 130 is the END statement and, consequently, the program stopped execution. Therefore, this program is flexible enough to be run for as many employees as you want. But to stop the program, just type the capital letter N.

IF..THEN Compound Statement

Some programming problems require more complex logical expressions. For example, we may want to execute a certain statement if both or one of the conditions is satisfied. That is, to test a range of values in order to execute either the statement following the THEN or the one following

the ELSE. These types of arrangements are referred to as compound conditionals. These statements require the use of compound operators as discussed in Chapter two. The general form of the compound conditional is:

```
                                       OR
 Ln1 IF relational expression             relational expression THEN Ln2
                                       AND
```

This statement consists of the statement number (Ln1), the keyword IF followed by an expression, and one of the compound operators OR or AND, another expression followed by the keyword THEN and a statement number (Ln2) or a BASIC statement. For example,

```
10 IF  G>90 AND G<100 THEN  PRINT " YOUR GRADE IS AN A"
```

In this example, if the grade of a student is between 90 and 100, the student receives an A grade (first division) in this course. However, these compound statements have many other uses. Specifically, in the United States, every income earner pays an income tax at a certain tax rate depending on the income of the taxpayer. Many computer programs are available for preparing taxes for taxpayers. For example, one of the statements could look like this:

```
10 IF  I>50000  AND  I<100000  THEN  T = I  * .30
```

In this statement, if the taxpayer earns somewhere between Rs. 50,000 and Rs. 100,000, the tax rate would be 30 percent. That is, this taxpayer would pay 30 percent of the income as tax.

On the other hand, the OR operator requires that if either of the conditions is true then execute the statement.

For example,

```
10 IF A>50 OR M$="MALE" THEN PRINT " PERSON IS EITHER OVER 50
                                    YEARS OLD OR A MALE"
```

In this statement, the age (A) or the sex (male) is checked. If the information provided shows that the person is over 50 years of age, the message after the word THEN will be printed. However, if an age of 50 or less is typed and the person was a female, the message will not be printed. The computer first checks the first logical relation, and if true then it executes the statement after the THEN. However, if the first condition is false, it then checks the second condition. If the second is true it then executes the statement after the THEN. However, if both conditions are false, then it executes the statement following this statement. A complete list of possibilities under which the message will or will not be printed is shown in the following table:

TABLE 5.2

IF ... OR .. THEN Condition

Value of A	Value of M$	Message
51 or over	male or female	will print the message
51 or over	male	will print the message
51 or over	female	will print the message
50 or less	male	will print the message
50 or less	female	will not print message

Let us modify Program 5.7 to print the letter grade of each student in addition to calculating an average grade of the ten students in a class. The program would look as follows:

PROGRAM 5.9

```
 10  T = 0
 20  T1 = 0
 30  PRINT " TYPE THE GRADE OF A STUDENT";
 40  INPUT G
 50  IF G>90 AND G<= 100 THEN PRINT " A"
 60  IF G>80 AND G<= 90  THEN PRINT " B"
 70  IF G>70 AND G<= 80  THEN PRINT " C"
 80  IF G>60 AND G<= 70  THEN PRINT " D"
 90  IF G<=60 THEN PRINT " E"
100  T = T + 1
110  T1 = T1 + G
120  IF T < 10 THEN 30
130  A1 = T1 / T
140  PRINT "THE AVERAGE GRADE OF "; T ;"  STUDENTS IS "; A1
150  END
```

As mentioned previously, to create a counter, you initialize the counter first. Lines 10 and 20 are still used for initializing variables T and T1. After initialization, line 100 is keeping a count of the number of students, and line 110 is adding up the grades as they are inputted. When the value of variable T exceeds 10, the next statement, line 140, is executed. Line 140 calculates the average. That is, it divides T1, the total of all grades, by T, by the number of students. Lines 50 through 90 are determining and print-

ing the letter grade based on the score of the student. That is, the grades are determined based on the following scale:

TABLE 5.3

Student Score and Grade

Score	Letter Grade
0 - 60	E
61 - 70	D
71 - 80	C
81 - 90	B
91 - 100	A

Therefore, a student with a score of 95 will fall between the range of 90 and 100 and will be given grade A, and so on. The output of the program is shown below:

OUTPUT 5.9

```
TYPE THE GRADE OF A STUDENT ? 66
D
TYPE THE GRADE OF A STUDENT ? 87
B
TYPE THE GRADE OF A STUDENT ? 85
B
TYPE THE GRADE OF A STUDENT ? 75
C
TYPE THE GRADE OF A STUDENT ? 95
A
TYPE THE GRADE OF A STUDENT ? 100
A
TYPE THE GRADE OF A STUDENT ? 88
B
TYPE THE GRADE OF A STUDENT ? 57
E
TYPE THE GRADE OF A STUDENT ? 46
E
TYPE THE GRADE OF A STUDENT ? 50
E
THE AVERAGE GRADE OF   10  STUDENTS IS   74.9
```

In the above output, when the first score of 66 is typed, the computer checks this value. In line 50, the condition is not true. That is, 66 is not between 90 and 100. It then checks line 60. The condition is still not true. It checks line 70. The condition is still not true. But, the condition is true in line 80, and it prints the letter grade D. However, line 90 will also be checked regardless of whether the computer finds a true condition or not. When the second score 88 is typed, the program starts checking all the conditions. It is not in the range specified in line 50, but line 90 is true and it prints the letter grade B. However, it will continue to check lines 70 through 90 even though it found a statement with the true condition.

One of the practical uses of the compound statement could be a schedule of loan payments. That is, we could always use a program which would calculate payments of a loan for a house, a car, or anything. As a borrower, all of us would like to know the amount of each monthly and how much of the payment goes to the principal and how much goes toward the interest. The following program answers these questions:

PROGRAM 5.10

```
 10 PRINT "TYPE THE AMOUNT OF LOAN";
 20 INPUT A1
 30 PRINT "TYPE INTEREST RATE";
 40 INPUT R1
 50 PRINT "TYPE NUMBER OF YEARS OF THE LOAN";
 60 INPUT Y1
 70 B1=A1
 80 F = 0
 90 I = 1
100 Y1= Y1 * 12
```

(continued on next page)

PROGRAM 5.10 (continued)

```
110 R1=R1/100
120 F=F+(1/((1+(R1/12))**I))
130 I=I+1
140 IF I<Y1 THEN 120
150 A = B1/F
160 PRINT
170 PRINT "YOUR MONTHLY PAYMENT IS ";A
180 PRINT
190 PRINT "DO YOU WANT TO LIST MONTHLY PAYMENTS";
200 INPUT H$
210 IF H$="NO" THEN 380
220 PRINT "HOW MANY YEARS DO YOU WANT TO PRINT";
230 INPUT H1
240 H1=H1*12
250 I=1
260 PRINT
270 PRINT "                    MONTHLY PAYMENT SCHEDULE"
280 PRINT "   BEG.                                          ENDING"
290 PRINT " BALANCE  INTEREST  PRINCIPAL AMOUNT   BALANCE"
300 C = B1*R1/12
310 PRINT USING "#######.##  ####.##    ####.##   #####.##
 #####.##";B1;C;(A-C);A;(B1-(A-C))
320 B1 = B1 - (A-C)
330 I=I+1
340 IF I<H1 THEN 300
350 END
```

In the above program, line 120 calculates the payment factor by using the following equation:

Factor = Factor + 1 /(1 + (interest rate/12)) ** month

Line 150 calculates the monthly payment by using the following equation:

Monthly payment = Loan amount / Factor

This calculates the amount one has to pay each month including interest to pay the loan over the life of the loan. The output of the program is as follows:

```
                    OUTPUT

   TYPE THE LOAN AMOUNT ? 1000
   TYPE INTEREST RATE ? 10
   TYPE THE NUMBER OF YEARS OF THE LOAN? 1

   YOUR MONTHLY PAYMENT IS :  95.52

   DO YOU WANT TO LIST MONTHLY PAYMENTS? YES
   HOW MANY YEARS YOU WANT TO PRINT ? 1

                 MONTHLY  PAYMENT SCHEDULE

   BEG.                                              ENDING
  BALANCE      INTEREST     PRINCIPAL    AMOUNT     BALANCE
 -----------------------------------------------------------
  1000.00        8.33         87.18      95.52      912.82
   912.82        7.61         87.91      95.52      824.91
   824.91        6.87         88.64      95.52      736.26
   736.26        6.14         89.38      95.52      646.88
   646.88        5.39         90.13      95.52      556.75
   556.75        4.64         90.88      95.52      465.88
   456.88        3.88         91.64      95.52      374.24
   374.24        3.12         92.40      95.52      281.84
   281.84        3.25         93.17      95.52      188.67
   188.67        1.57         93.95      95.25       94.73
    94.73        0.79         94.73      95.52        0.00
```

As you can see in the output, one has to pay Rs. 95.52 per month which includes interest and principal. The principal amount reduces the loan each month. Each month the interest decreases and the amount of payment towards the principal increases. During this period, enough payments are made to cover the Rs. 1000 loan as well as interest on the loan.

QUESTIONS

1. Correct errors, if any, in the following program segments:

a)
```
20 GOTO 100
30 PRINT "NAME"
40 END
```

b)
```
20 GO TO 20
30 PRINT "NAME"
40 END
```

c)
```
20 GO TO PRINT
30 PRINT "NAME"
40 END
```

d)
```
20 GO TO 40
30 PRINT "NAME"
40 END
```

2. What is an abnormal termination of a program? When would abnormal termination occur.

3. What is structured programming? Discuss the advantages of structured programming.

4. Print the output of the following program segments:

a)
```
10  A = 10
20  IF  10 = A THEN PRINT "A IS EQUAL TO 10"
30  A = 20
40  GO TO 10
```

b)
```
10  A = 10
20  IF 10 = A THEN PRINT A
30  GO TO 60
40  A = 30
50  IF A = 40 THEN PRINT A
60  END
```

5. What is a sentinel record? When would you use this record?

6. What is a counter and what is it used for?

7. Show the output of the following program segments:

a)
```
10  A = 0
20  A = A + 1
30  PRINT A
40  IF A = 10 THEN END
50  GO TO 20
```

b)
```
10  C = 0
20  C = C + 2
30  PRINT C
40  C = C + 1
50  IF C = 40 THEN END
60  GO TO 20
```

8. Write a program that reads and prints grades, one for each student, of ten students and calculates the average grade for the class.

9. Write a program that accepts the salesperson's name and the amount sold and prints the name, the amount sold, and the commission paid to the salesperson based on the following table:

AMOUNT SOLD	COMMISSION RATE
0 - 1000	0 %
1001 - 5000	2 %
5001 - 10000	5 %
10001 - 20000	10 %
20000 and above	15 %

10. Write a program that determines overdue accounts based on the following table of a company's account receivables:

DAYS LATE	MESSAGE
0 - 30	None
31 - 60	Overdue 30 days
60 - 120	Overdue 60 days, you must pay promptly
120 - 150	Overdue, you must pay within 10 days.
150 and over	Delinquent. Take legal action.

11. Write a program that converts and prints the following temperatures in Fahrenheit (F) into Celsius, and coverts them back into Fahrenheit. The equation for converting Fahrenheit into Celsius is:

$$C = 5/9 \times (F - 32)$$

Temperatures

60, 80, 90, 100, 105, 110, 120, 66, 55, 88, 75

12. Write a program that prints numbers between 10 and 20 inclusive. You must use counter and must not read or input the numbers into the program.

13. Write a program that reads the height and weight of a man and determines whether he is overweight, underweight, or average weight. Use the following table:

Height	Average Weight
5' - 1"	128 - 134
5' - 2"	134 - 140
5' - 3"	140 - 144
5' - 4"	143 - 149
5' - 5"	147 - 153
5' - 6"	152 - 158
5' - 7"	156 - 163
5' - 8"	161 - 167
5' - 9"	166 - 172
5' - 10"	171 - 176

5' - 11"	175 - 181
6' - 0"	181 - 186
6' - 1"	186 - 191
6' - 2"	191 - 196
6' - 3"	197 - 200

14. Write a program that reads the sex and grade of each student and prints the grade and the sex of the student. (Hint: you can code the sex of a student when inputting it into the program. For example, female could be coded as 1 and male could be coded as 0.)

15. Write a program that reads the wages of employees and deducts income tax based on the following table.

Wages	Tax Rate
Rs. 0 - 10000	0 %
10001 - 20000	5 %
20001 - 30000	7 %
30001 - 50000	9 %
50001 - 90000	15 %
90001 - 150000	20 %
150000 and over	30 %

16. The population of Pakistan was reported to be 100,000,000 in 1988, and was growing at the rate of 3.25 percent per year. Write a program that will project the population into the year 2020. However, if the population exceeds 120,000,000 in the year 2000, use a 2.00 percent growth rate from there on.

CHAPTER 6

STRUCTURED PROGRAMMING : FOR - NEXT STATEMENT

FOR...NEXT LOOP

So far we have seen several different patterns of loops. In all patterns a condition is checked against the specified value to end the loop. The loop is terminated if the condition is satisfied. In some cases, the conditional statement is placed in the beginning of the loop, called **pretest loop**. In other cases, the conditional statement is placed at the end of the loop, called **posttest loop.** All the loops were discussed in the context of IF ... THEN or GOTO statements. However, a loop may have other loops within it, called **nested loops**. That is, loop may have been nested in another processing looping such as another IF...THEN or GOTO loop. When there is a loop within a loop, the most encompassing loop is called the outer loop and the loop which is inside this loop is referred to as the inner loop. In such a set up, the inner loop is executed repeatedly as many times as required by the inner loop each time the outer loop is executed. The concept of nested loop is applicable in all looping techniques. However, it is more appropriate in the context of the FOR ... NEXT loop. For example, in Program 5.6, the counter Y was initialized in line 40, incremented in line 100, and tested in line 120. Because of the importance of the counting loop and nested loop, BASIC has provided the FOR ... NEXT statements for the counting loop as a control.

The general form of the FOR ... NEXT is as follows:

```
                          number      number       number
Ln1 FOR varname1 =          or   TO     or STEP      or
                        varname2    varname3     varname4
                                                    >Increment value
       .                                            >Limit value
       .                                            >Initial value
       .                                            >Control variable
   BASIC Statements to be repeated
       .
       .

 Ln2 NEXT varname1
```

In a program, the two statements FOR and NEXT must appear together and Ln1 comes first in the sequence of statement numbers. That is, the FOR statement must come before the NEXT statement, and each FOR must have a corresponding NEXT. A FOR loop begins with a FOR statement, contains any number of BASIC statements, and ends with a NEXT statement. The varname1, a numeric variable, called an **index** or loop counter, must be the same in both statements. In the FOR statement, after the equal sign, there could be three numeric constants or three numeric variables that are previously defined in the program outside of the FOR ... NEXT statements. However, the STEP and the varname4 or constant after the STEP are not required, if the value is 1. The three numbers or variables represent the initial value, the test value, and the step or increment in that order. They determine the number of times a loop is executed.

As an example:

PROGRAM 6.1

```
10 FOR I = 1 TO 10  STEP 1
20        PRINT I;
30 NEXT I
40 END
```

In the above program, the FOR statement is in line number 10. After the line number, the word FOR starts the statement. The index variable is I which is the same for the NEXT statement in line number 30. When the FOR ... NEXT starts at line 10, the index I is initialized to 1, because the value after the equal sign equals 1. The statements within the loop, lines 20 through 30, are executed without testing the limit value of the loop. Therefore, the next statement number 20, PRINT I, is executed. That is, it prints the number 1, the initial value of I. Then it goes to statement number 30, the NEXT statement. The NEXT statement indicates the end of the loop. Upon execution of the NEXT statement, the loop goes on back to statement number 10, the FOR statement. Since it is the second time the FOR statement is executed, the value of the variable I is incremented by 1, the value next to the STEP. Now the value of the variable I is 2. The value of the variable I is tested against the limit value, 10. Since the index is less than the limit value, the loop is repeated. That is, the value of I is printed as 2. Each time the loop is executed, I is increased by 1 and tested with the limit value until the value of I exceeds the limit value, 10. Then the computer stops repeating the loop and goes on to the statement after the NEXT statement, the END statement in this program. This loop will be executed 10 times. That is, the PRINT statement will print values from 1 to 10. The output of the program would look as follows:

OUTPUT 6.1

```
1  2  3  4  5  6  7  8  9  10
```

Let us consider another example:

```
          PROGRAM 6.2

10    FOR I  =   1   TO  10   STEP   5
20                 PRINT  I;
30    NEXT I
40    END
```

In this example, the value after the STEP is now 5. The program will initial the I to 1. The PRINT statement will print the number 1 as the value of I. The second time, the value of I will be increased by 5. The value of I will now be 6. The PRINT statement will print 6. The third time, the value of I will be increased by 5. The I will become 11. Since the value of I is greater than the limit value, the loop will not be executed. The computer will go on to statement number 40, the END statement and the program will stop. The output of the program is as follow:

```
          OUTPUT 6.2

  1  6
```

If the increment is 1, the STEP is not needed. In a case where a STEP is not specified, the computer automati-

cally increments the index by 1, a **default value.** This automatic increment procedure is called **default**. It means that the computer is programmed to perform some activities if not specified by a programmer. Therefore, in Program 6.1, statement 10 could be written as:

```
10    FOR  I  =  1  TO   10
```

In this statement, the STEP is not specified. The computer will use the default procedure to increment the value of I by 1.

Further, the initial value of an index, the limit value, and the increment value could be any value. That is, they could be negative, zero (except the increment value should not be zero) or positive. For example:

```
10    FOR  I  =  5   TO  10
```

In this statement, the index is initialized to 5, incremented by 1, a default value because the STEP component is not specified, and the limit value is 10. Output of this statement is:

```
5  6  7  8  9  10
```

The FOR ... NEXT values could be fractional. For example:

```
10    FOR  I = .01 TO 1 STEP .1
```

In this statement, the index is initialized to .01, incremented by .1, and the limit value is 1. The output of this statement is as follows:

```
.01  .11  .21  .31  .41  .51  .61  .71  .81  .91
```

The FOR ... NEXT statements could be used to count backwards. For example:

```
10   FOR   I  =  10  TO  1   STEP   -1
```

Notice that in the backward count, the initial value is a larger number that the limit value, and the increment value is always negative. The output of this statement is:

```
10   9   8   7   6   5   4   3   2   1
```

Besides constants, variables could also be used. For example,

PROGRAM 6.3

```
10   K = 1
20   M = 10
30   FOR I = K TO M
40             PRINT I;
50   NEXT I
60   END
```

In this example, the initial variable K and the limit variable M are defined in statements numbered 10 and 20 respectively. The FOR statement consists of variable names instead of constants.

In addition to using variables, expression can be formed by using variables and/or constants. For example,

PROGRAM 6.4

```
10    K = 1
20    M = 10
30    FOR I = K TO M*2 STEP M/2
40              PRINT I;
50    NEXT I
60    END
```

In this example, the initial variable K and the limit variable M are defined in statements numbered 10 and 20 respectively. The FOR statement consists of expressions, M*2 and M/2. This means that the I will be initialed to 1, incremented by 5 (M/2 = 10/2), and tested with the limit value of 20 (M*2 = 10*2).

However, there are some conditions which you should watch for.

A STEP value which puts the index beyond the limit value after the first execution of the loop does not serve any function. For example,

```
10  FOR  I  =  1  TO  10   STEP   10
```

In this statement, the loop will be executed only once. The second time, the index is incremented by 10, I equals 11 which is greater than 10. The loop is terminated. This loop is executed only once; therefore, it does not serve any purpose.

Also, one must be cautious about branching into the loop from outside of the loop. That is, it is important never to branch into the loop from outside of the loop without executing the FOR statement first. For example,

PROGRAM 6.5

```
10   GO TO 50
20   PRINT " EXAMPLE WITH INVALID USE OF FOR ... NEXT"
30   K = 10
40   FOR I = 1 TO  K
50          PRINT I;
60   NEXT I
70   END
```

In this example, statement number 10, GOTO 50, goes to statement number 50, which is inside the FOR ... NEXT loop. Since the program never executed line 40, the FOR statement, the computer will assume that there is no FOR statement in the program. An error message will be issued.

However, it is permissible to branch out of the FOR ... NEXT loop. But once out of the loop, you can reenter only by executing the FOR statement first. That is, to execute the FOR ... NEXT loop, you must execute the FOR statement first.

Other examples of errors are:

TABLE 6.1

EXAMPLE OF FOR ... NEXT ERROR

BASIC Statement	Valid/Invalid	Reason
10 FOR I = 1 TO 10 20 I = 1 30 NEXT I	Invalid	The index variable I is redefined within the FOR ... NEXT loop
10 FOR J = 1 TO 1 : 40 NEXT J	Valid	Redundant use of FOR ... NEXT. It will execute the loop once.
10 FOR J = 1 TO 10 : 40 NEXT A	Invalid	The index variable A in the NEXT statement does not match the inc variable J in the FOR statement.
10 FOR R = 10 TO 1 . . 40 NEXT R	Invalid	The increment value must be specified in a backward count
10 FOR R = 1 TO 10 . . 60 END	Invalid	The NEXT statement is missing. The progra will issue an error message: FOR without NEXT in line 10.
10 K = 1 . . 40 NEXT R	Invalid	The FOR statement is missing. The progra will issue an error message: NEXT without FOR in line 40.

Keep in mind that when the value of I is defined, it can be used later in the program. That is, the index is like other variables in the program; once defined it stays defined during the execution of the program unless changed in the other parts of the program. For example:

PROGRAM 6.6

```
10  FOR I   =  1  TO   10
20       PRINT  I;
30  NEXT I
40  PRINT " THE CURRENT VALUE OF I IS "; I
50   I = 1
60  PRINT
70  PRINT " THE NEW VALUE OF I IS "; I
80  END
```

In this program the value of I will be 11 when printed by statement number 40, and then changed to 1 by statement number 50. The output of the program is as follows:

OUTPUT 6.3

```
THE CURRENT VALUE OF I IS   11
THE NEW VALUE OF I IS  1
```

Beside controlling the loop, the FOR ... NEXT can be used to generate numbers. For example, let us write a program that generates even numbers:

PROGRAM 6.7

```
10   PRINT " FOLLOWING IS THE LIST OF EVEN NUMBERS"
20   PRINT " BETWEEN 1 TO 50."
30   FOR J = 2  TO  50   STEP  2
40             PRINT J;
50   NEXT J
60   END
```

This program generates a list of the following even numbers between 1 and 50.

OUTPUT 6.4

```
FOLLOWING IS THE LIST OF EVEN NUMBERS
BETWEEN 1 and 50

2  4  6  8  10  12  14  16  18  20  22  24  26  28  30  32
34  36  38  40  42  44  46  48  50
```

With the FOR ... NEXT statements, we can generate numbers and use those numbers for other purpose. For example, the following program calculates the squares of odd numbers between 1 and 10:

PROGRAM 6.8

```
10  PRINT " FOLLOWING IS A .LIST OF ODD NUMBERS"
20  PRINT " BETWEEN 1 TO 10 AND THEIR SQUARES."
30  PRINT "NUMBER      SQUARE "
40  PRINT "---------------------"
50  FOR J = 1  TO  10   STEP  2
60            PRINT J;"            ";J*J
70  NEXT J
80  END
```

This program generates a list of the following odd numbers between 1 to 10, and their squares.

OUTPUT 6.5

```
FOLLOWING IS THE LIST OF ODD NUMBERS
BETWEEN 1 TO 10 AND THEIR SQUARES.

NUMBER        SQUARED
---------------------
 1              1
 3              9
 5             25
 7             49
 9             81
```

The implementation of the counting loop is simplified by the FOR ... NEXT statements. In some cases, the FOR ... NEXT statement shortens the program. That is, it takes fewer programming statements to write the same program as in the IF..THEN. For example, Program 5.6 would look as follows with the FOR...NEXT statements:

PROGRAM 6.9

```
 10  PRINT "          INTEREST      "
 20  PRINT "YEAR      AMOUNT     BALANCE"
 30  PRINT "---------------------------"
 40  FOR  Y =  1  TO  5
 50  READ P,I
 60  A = P
 70  A = A * (1 + I) ** Y
 80  B = A - P
 90  PRINT USING;"###      #####.##      ######.##";Y;B;A
100  P = A
110  NEXT Y
120  DATA 1000,.10
130  END
```

In this program, one less statement is needed than in Program 5.6.

In addition, Program 5.7, the grade program, could be rewritten as follows:

PROGRAM 6.10

```
10  T1 = 0
20  FOR  T =  1   TO   10
30  PRINT " TYPE THE GRADE OF A STUDENT";
40  INPUT G
50  T1 = T1 + G
60  NEXT T
70  A1 = T1 / (T - 1)
80  PRINT "THE AVERAGE GRADE OF "; (T-1);"  STUDENTS IS "; A1
90  END
```

In this program, we now need one counter for summing the grades. The index, T , provides for the total number of students. However, when the index is terminated, the value of the index will be 11. Therefore, we need to subtract 1 from T to make the index equal to 10. Statements 70 and 80 were changed to adjust the index before dividing the grade total by the student's total and printing the student's total.

The output of the program will look as follows:

OUTPUT 6.6

```
TYPE THE GRADE OF A STUDENT ? 66
TYPE THE GRADE OF A STUDENT ? 87
TYPE THE GRADE OF A STUDENT ? 85
TYPE THE GRADE OF A STUDENT ? 75
TYPE THE GRADE OF A STUDENT ? 95
TYPE THE GRADE OF A STUDENT ? 100
TYPE THE GRADE OF A STUDENT ? 88
TYPE THE GRADE OF A STUDENT ? 57
TYPE THE GRADE OF A STUDENT ? 46
TYPE THE GRADE OF A STUDENT ? 50
THE AVERAGE GRADE OF   10  STUDENTS IS   74.9
```

As discussed in chapter 5, one of the practical examples of the compound statement could be a schedule of loan payments. That is, we could always use a program which would calculate payment of a loan for a house, a car, or anything. As borrowers, all of us would like to know the amount of our monthly payments and how much of the payments go to principal and how much towards interest. The following modified program answers the same questions as in Program 5.10:

PROGRAM 6.11

```
 10 PRINT "TYPE THE AMOUNT OF LOAN";
 20 INPUT A1
 30 PRINT "TYPE INTEREST RATE";
 40 INPUT R1
 50 PRINT "TYPE NUMBER OF YEARS OF THE LOAN";
 60 INPUT Y1
 70 B1=A1
 80 F = 0
 90 Y1= Y1 * 12
100 R1=R1/100
110 FOR  I = 1 TO  Y1
120 F=F+(1/((1+(R1/12))**I))
130 NEXT I
140 A = B1/F
150 PRINT
160 PRINT "YOUR MONTHLY PAYMENT IS ";A
170 PRINT
180 PRINT "DO YOU WANT TO LIST MONTHLY PAYMENTS";
190 INPUT H$
200 IF H$="NO" THEN 380
210 PRINT "HOW MANY YEARS DO YOU WANT TO PRINT";
220 INPUT H1
230 H1=H1*12
```

(continued on next page)

PROGRAM 6.11 (continued)

```
240 PRINT
250 PRINT "                    MONTHLY PAYMENT SCHEDULE"
260 PRINT "   BEG.                                          ENDING"
270 PRINT " BALANCE  INTEREST  PRINCIPAL AMOUNT   BALANCE"
280 FOR I = 1  TO  H1
290 C = B1*R1/12
300 PRINT USING "####### ##  ####.##    ####.##   #####.##
#####.##";B1;C;(A-C);A;(B1-(A-C))
310 B1 = B1 - (A-C)
320 NEXT I
330 END
```

In the above program, the counter I was replaced by the FOR ... NEXT statement. Now, we have two fewer statements than in Program 5.10. The output of the program will not be affected. The output is as follows:

OUTPUT 6.7

```
TYPE THE LOAN AMOUNT ? 1000
TYPE INTEREST RATE ? 10
TYPE THE NUMBER OF YEARS OF THE LOAN? 1

YOUR MONTHLY PAYMENT IS :  87.92

DO YOU WANT TO LIST MONTHLY PAYMENTS? YES
HOW MANY YEARS YOU WANT TO PRINT ? 1
```

(continued on next page)

OUTPUT 6.7 (continued)

MONTHLY PAYMENT SCHEDULE

BEG. BALANCE	INTEREST	PRINCIPAL	AMOUNT	ENDING BALANCE
1000.00	8.33	79.58	87.92	920.42
920.42	7.67	80.25	87.92	840.17
840.17	7.00	80.91	87.92	759.26
759.26	6.33	81.59	87.92	677.67
677.67	5.65	82.27	87.92	595.40
595.40	4.96	82.95	87.92	512.45
512.45	4.27	83.65	87.92	428.80
428.80	3.57	84.34	87.92	344.46
344.46	2.87	85.05	87.92	259.41
259.41	2.16	85.75	87.92	173.66
173.66	1.45	86.47	87.92	87.19
87.19	.73	87.19	87.92	0.00

As you can see, there is no change in the output.

NESTED LOOP - FOR...NEXT STATEMENTS

The FOR...NEXT statements can also be used to create a **nested loop**. For example,

PROGRAM 6.12

```
10  PRINT " I  J"
20  FOR  I =  1   TO   2
30     FOR  J =  1  TO  3
40        PRINT I;J
50     NEXT J
60  NEXT I
70  END
```

In this program, the FOR...NEXT loop in lines 30 through 50 is nested within the FOR ... NEXT loop in lines 20 and 60. Nesting loops require that each FOR statement must have a matching NEXT statement. Further, the paired FOR ... NEXT statements should have the same control variables. In our program, the paired FOR..NEXT statements in lines 30 and 50 have the variable J as the control variable, and the paired FOR ... NEXT statements in 20 and 60 have the variable I as the control variable. The inner loop is always paired first before the outer loop and the inner loops are executed first before the outer loop. In the nested loop, the outer loop is initialized and executed first. The program continues and sees the inner loop. The loop is initialized and executed as many times as specified by the limit value, three times in our example. Then the program goes on to the next statement, the NEXT statement. Since it is a part of the outer loop, the program goes on to line 20, the FOR statement. The I variable is incremented to 2 and the next statement is executed, the FOR statement in line 30. Since the FOR statement was executed from the preceding statement in the program, the J is again initialized to 1, and the inner loop starts over. The inner loop will be executed as many times as specified in the limit value. After executing the inner loop, the NEXT statement following the inner loop will be executed. The loop then returns to the FOR statement. The value of I is incremented by 1. This time the value of I is greater than the limit value, and the execution of the outer loop stops. The output of the program would look as follows:

```
OUTPUT  6.8

  I  J

  1  1
  1  2
  1  3
  2  1
  2  2
  2  3
```

The above output will help in understanding how the nested loops work. First the FOR statement in line 20 is executed. At this time, the value of I is initialized to 1. Then the inner loop is found and initialized to 1. The PRINT statement prints 1 for both control variables I and J. Since the inner loop is executed first, the value of I is fixed at 1 while the value of J changes. After the inner loop is executed three times, the limit value, the computer goes on to the next statement, line 60. This line is the NEXT statement paired with the FOR statement in line 20. The computer goes back to line 20 and increments the value of I by 1. The value of I is now 2. The execution continues with the execution of the FOR statement in line 30, executing the inner loop 3 times. The PRINT statement will print three lines of output again with I fixed at 2 this time. After the inner loop is executed, the process continues until the control variable of the outer loop exceeds its limit value.

Let us take an example with three nested loops:

PROGRAM 6.13

```
10    PRINT  " I       J       K"
20    FOR  I  =  1  TO  2
30       FOR  J = 1   TO   3
40          FOR   K   =  1   TO   4
50      PRINT I;"     ";J;"     ";K
60          NEXT  K
70       NEXT J
80    NEXT I
90    END
```

In this program there are three loops: statements 40 and 60, 30 and 70, and 20 and 80. All these are matching FOR...NEXT statements. The FOR...NEXT with control variables K is the inner loop, the one with J is the outer of this loop, and the one with I is the extreme outer loop. The

output of the program is as follows:

OUTPUT 6.9

I	J	K
1	1	1
1	1	2
1	1	3
1	1	4
1	2	1
1	2	2
1	2	3
1	2	4
1	3	1
1	3	2
1	3	3
1	3	4
2	1	1
2	1	2
2	1	3
2	1	4
2	2	1
2	2	2
2	2	3
2	2	4
2	3	1
2	3	2
2	3	3
2	3	4

In this output, the innermost loop, the one with the control variable K, is executed six times. The loop with the control variable J is executed twice, and the loop with the control variable I is executed twice.

In order to determine how many times the nested loop will be executed, multiply the limit values of all the FOR...NEXT statements. In our example, the nested loop will be executed 24 times (2 x 3 x 4).

There is no limit to the number of level loops that can be nested, as long as they do not cross (overlap) each other. Examples of valid or invalid loops could be:

TABLE 6.2

EXAMPLES OF VALID AND INVALID NESTED LOOPS

```
        VALID                                INVALID
---------------------------------------------------------------
 FOR   I =  1  TO  5              FOR  I  = 1  TO  5
 FOR   J =  1  TO  10             FOR  J  = 1  TO  10
      ...                              ...
      ...                              ...
 NEXT  J                          NEXT I
 NEXT  J                          NEXT J

 FOR   I =  1  TO  5              FOR  I  = 1  TO  5
 FOR   J =  1  TO  10             FOR  J  = 1  TO  10
 FOR   K =  1  TO  5              FOR  K  = 1  TO  5
         ...                            ...
         ...                            ...
 NEXT  K                          NEXT J
 NEXT  J                          NEXT K
 NEXT  I                          NEXT I
---------------------------------------------------------------
```

One of the practical example of nested loops is creating a table with row and column labels. For example,

PROGRAM 6.14

```
 10  PRINT  "                    TABLE   1"
 20  PRINT
 30  FOR I = 1  TO  8
 40     PRINT "   ";I;
 50  NEXT I
 60  PRINT
 80  FOR I  = 1 TO  9
 90  FOR K  = 1 TO  8
100   PRINT "--------";
110  NEXT K
120  PRINT
130    PRINT I;
140     FOR J = 1 TO 8
150     PRINT "|       ";
160     NEXT J
170     PRINT
180  NEXT I
190  END
```

The output of the above program is:

OUTPUT 6.11

TABLE 1

```
 1        2        3        4        5        6        7        8
-------------------------------------------------------------------
1 |        |        |        |        |        |        |        |
-------------------------------------------------------------------
2 |        |        |        |        |        |        |        |
-------------------------------------------------------------------
3 |        |        |        |        |        |        |        |
-------------------------------------------------------------------
4 |        |        |        |        |        |        |        |
-------------------------------------------------------------------
5 |        |        |        |        |        |        |        |
-------------------------------------------------------------------
6 |        |        |        |        |        |        |        |
-------------------------------------------------------------------
7 |        |        |        |        |        |        |        |
-------------------------------------------------------------------
8 |        |        |        |        |        |        |        |
-------------------------------------------------------------------
9 |        |        |        |        |        |        |        |
-------------------------------------------------------------------
```

This matrix table is printed by the Program 6.14. By understanding the nested loops, one can find many useful applications for them. Some of the uses will be discussed in the next chapter.

Let us modify the Grade program to allow for three examination grades for each student and print the average grade for each student and the average grade for the class.

For example:

PROGRAM 6.15

```
 10 T1 = 0
 20 FOR I = 1 TO 10
 30 PRINT " TYPE THREE EXAMINATION GRADES FOR EACH STUDENT"
 40 T = 0
 50   FOR K = 1 TO 3
 60     INPUT G
 70     T  = T + G
 80   NEXT K
 90 A2 = T / 3
100 PRINT " AVERAGE GRADE OF THIS STUDENT IS "; A2
110 T1 = T1 + A2
120 NEXT I
130 A1 = T1 / (I-1)
140 PRINT" THE AVERAGE GRADE OF THE"; (I-1)"; STUDENTS IS "; A1
150 END
```

In this program, we now need one counter for summing the grade of each student and one counter for the whole class. The index I provides the total number of students, and counters T and T1 are used for summing grades. In addition, we need two loops. One to sum and enter three grades for each student and one to enter grades for ten students. In the latter case, when the index is terminated, the value of the index will be 11. Therefore, we need to subtract 1 from I to make the index equal to 10. Statements 130 and 140 were changed to adjust the index before dividing the grade total by the student's total, and printing the students total.

The output of the program will look as follows:

OUTPUT 6.11

```
 TYPE THREE EXAMINATION GRADES FOR EACH STUDENT
? 60
? 80
? 90
 AVERAGE GRADE OF THE STUDENT IS 76.6666
 TYPE THREE EXAMINATION GRADES FOR EACH STUDENT
? 50
? 60
? 60
 AVERAGE GRADE OF THE STUDENT IS 56.6666
 TYPE THREE EXAMINATION GRADES FOR EACH STUDENT
? 90
? 95
? 98
 AVERAGE GRADE OF THE STUDENT IS 94.3333
 TYPE THREE EXAMINATION GRADES FOR EACH STUDENT
? 88
? 87
? 81
 AVERAGE GRADE OF THE STUDENT IS 85.3333
 TYPE THREE EXAMINATION GRADES FOR EACH STUDENT
? 76
? 76
? 77
 AVERAGE GRADE OF THE STUDENT IS 76.3333
 TYPE THREE EXAMINATION GRADES FOR EACH STUDENT
? 89
? 95
? 77
```

(continued on the next page)

```
 AVERAGE GRADE OF THE STUDENT IS 87
 TYPE THREE EXAMINATION GRADES FOR EACH STUDENT
? 45
? 65
? 55
 AVERAGE GRADE OF THE STUDENT IS 55
 TYPE THREE EXAMINATION GRADES FOR EACH STUDENT
? 66
? 80
? 86
 AVERAGE GRADE OF THE STUDENT IS 77.3333
 TYPE THREE EXAMINATION GRADES FOR EACH STUDENT
? 90
? 80
? 88
 AVERAGE GRADE OF THE STUDENT IS 86
 TYPE THREE EXAMINATION GRADES FOR EACH STUDENT
? 50
? 50
? 40
 AVERAGE GRADE OF THE STUDENT IS 46.6666
THE AVERAGE GRADE OF   10  STUDENTS IS   74.1333
```

QUESTIONS

1. Define pretest, posttest, and nested loops.

2. What does an index do in the FOR... NEXT loop?

3. What would happen if the STEP value were not used in the FOR ... NEXT loop?

4. Write a program that reads and prints grades, one for each student, of ten students and calculates the average grade for the class.

5. Write a program that prints numbers between 10 and 20 inclusive. You must use counter and must not read or input the numbers into the program.

6. Write a program that reads the height and weight of a man and determines whether he is overweight, underweight, or average weight. Use the Do loop to generate the following table values.

Height	Average Weight
5' - 1"	128 - 134
5' - 2"	134 - 140
5' - 3"	140 - 144
5' - 4"	143 - 149
5' - 5"	147 - 153
5' - 6"	152 - 158
5' - 7"	156 - 163
5' - 8"	161 - 167
5' - 9"	166 - 172
5' - 10"	171 - 176
5' - 11"	175 - 181
6' - 0"	181 - 186
6' - 1"	186 - 191
6' - 2"	191 - 196
6' - 3"	197 - 200

7. Write a program that reads the sex and grade of each student and prints the grade and the sex of the student. (Hint: you can code the sex of a student when inputting it into the program. For example, female could be coded as 1 and male could be coded as 0.). Use DO loop.

8. Write a program that reads the wages of employees and deducts income tax based on the following table. Use the Do loop to create the table.

Wages	Tax Rate
Rs. 0 - 10000	0 %
10001 - 20000	2 %
20001 - 30000	4 %
30001 - 40000	6 %
40001 - 50000	8 %
60001 - 70000	10 %
80000 - 90000	12 %
100000 and over	14 %

9. Write a FOR...NEXT loop that prints the numbers 20, 18, 16,..., 0 in a column.

10. Show the number of lines and the values that will be printed by each of the following program segments.

a)
```
FOR I = 1 TO 5
PRINT I
NEXT I
```

b)
```
FOR I = 1 TO 20 STEP 5
PRINT I
NEXT I
```

c)
```
FOR I = 1 TO 10 STEP 10
PRINT I
NEXT I
```

d)
```
FOR J = 1 TO 2
FOR K = 1 TO 3
PRINT J,K
NEXT K
NEXT J
```

e)
```
FOR L = 10 TO 0 STEP -2
PRINT L
NEXT L
```

11. Correct errors, if any, in the following program segments:

a)
```
FOR  I = 10 TO 30  STEP -1
PRINT I
NEXT I
```

b)
```
FOR J = 1 TO 10
PRINT J
NEXT I
```

d)
```
FOR I = 10 TO 0 STEP 2
PRINT I
NEXT I
```

e)
```
FOR J = 1 TO 10
J = 1
PRINT J
NEXT J
```

c)
```
FOR L = 1 TO 10
FOR K = 1 TO 2
FOR J = 1 TO 3
PRINT L,K, J
NEXT J
NEXT L
NEXT K
```

12. Write a program that prints two columns of numbers. The first column should have the numbers between 10 and 20, and the second column between 40 and 60.

13. Write a program that will read the following table of numbers and will print the average of each row and each column.

10	20	44	45	50	80	90	50
60	30	66	43	25	29	40	60
40	50	60	40	30	20	34	43
35	99	20	39	50	60	77	88

14. A factorial of a number is calculated as follows:

$$n! = 3! = 3 \times 2 \times 1 = 6$$

Write a program that calculates a factorial of any number.

CHAPTER 7

ARRAY AND MATRIX - DIMENSION STATEMENT

INTRODUCTION

So far we inputted or read numbers. Most often a number was stored temporarily in the computer memory under a variable name and was replaced with another number. However, it is necessary to store and process a large amount of data. For example, to calculate an average and a variance of five numbers will require creating five separate variable names and accessing each one individually. Let us create a program to calculate the average and the variance of the following five numbers:

50, 70, 85, 99, and 65.

The equations for calculating an average and a variance are:

$$1) \qquad \bar{Y} = \frac{\Sigma Y}{n} = \frac{\text{sum of Y's}}{\text{number of numbers}} = \text{average}$$

$$\text{i.e.} \quad \bar{Y} = \frac{50 + 70 + 85 + 99 + 65}{5} = \frac{369}{5} = 73.8$$

$$2) \qquad \sigma = \frac{\Sigma\,(Y - \bar{Y})^2}{n - 1} = \text{variance}$$

$$= \frac{(50-73.8)^2 + (70-73.8)^2 + (85-73.8)^2 + (99-73.8)^2 + (65-73.8)^2}{4} = 354.$$

Equation 1 could be calculated without storing each number in separate memory (variable name). But equation 2 requires that we calculate the average first and then subtract the average from each number and square the difference. That means that either we reread the number again or store each one in a memory and access it later on. For example, the following program calculates both the average and the variance:

PROGRAM 7.1

```
10 READ J1,J2,J3,J4,J5
20 T = J1 + J2 + J3 + J4 + J5
30 A1= T / 5
40 V1=((A1-J1)^2+(A1-J2)^2+(A1-J3)^2+(A1-J4)^2+(A1-J5)^2))/(5-1)
50 PRINT "AVERAGE IS "; A1," VARIANCE IS "; V1
60 DATA 50,70,85,99,65
70 END
```

In this program, the READ statement reads the five numbers and stores them in the variable names J1, J2, J3, J4, and J5. After the computer executes the program, the computer memory would look as follows:

COMPUTER MEMORY

MEMORY LOCATION LABELS	J1	J2	J3	J4	J5	T	A1	V1
VALUES	50	70	85	99	65	365	73.8	354.7

The computer creates a memory space for each variable and puts the associated value in the memory. Each variable can hold **one** value. The computer knows the value by looking for the variable name in the memory. Therefore, when the program later refers to J1, the computer checks memory J1 and picks the number 50 out of the memory. Therefore, with the techniques discussed so far, we have to use a separate variable for each number in the list. That is, if we have 60 numbers, we would need 60 variable names for the data. This procedure makes it difficult for a programmer to name 60 variables and then refer to them in a equation as that in line 20 and 40. In addition, in other problems to be discussed later, even to refer to each variable will not be sufficient to solve the problem.

In this chapter we are going to learn about variables that can hold more than one value at a time. These variables are called **array.** Under this approach we can name the entire data list by a single variable. An array variable can hold more than one value at a time. The individual value in the list is then identified by referring to it by its position in the list. The name that refers to an array is called **array variable.** The naming rules of array variables are the same as a simple variable. Each value in an array is referred to as an **array element**. Each element has a location number in the array. That is, the first value in the array is the first element of the array and will be referred to by the location number in the array. Thus, if we were to use an array for the five numbers listed above, the memory would look as follows:

COMPUTER MEMORY ALLOCATED TO ARRAY J

	ARRAY VARIABLE J					
MEMORY LOCATION NUMBER	1	2	3	4	5	SUBSCRIPT
VALUES	50	70	85	99	65	

In this memory, the array variable, J, has location numbers. That is, J1 is now J(1), J2 is now J(2), and so on. Therefore, when we wish to refer to the value in the first location, we have to give the name and the location of the array, i.e. J(1). The computer keeps track of the elements by their numbers. In the context of an array these numbers are referred to as **subscripts**. It is important not to confuse a subscript with the value that is stored in the array. A subscript refers to one of the locations of the array, while value is the element (value) stored in that location. Thus, J(1) means that you want the first, the subscript, memory location of the array J. However, the array element 50 is stored there.

The data in an array is stored under one array variable name. However, in most cases, except in the matrix manipulation, an array variable cannot be used by itself collectively. That is, each array variable must be followed by its subscript to identify the element in the array. That is, by using a subscript, an element can be used as a simple variable.

SUBSCRIPT

A subscript identifies the element of an array. A subscript must be numeric variable, numeric constant, or numeric expression. That is, any of the following are valid subscripts:

```
10  PRINT  J(1)
```

or

```
10  K  =  1
20  PRINT J(K)

    or

10  K  =  1
20  L  =  2
30  PRINT J(K*L)
```

These statements show that a subscript could be created by using different forms. However, the most common form of subscript used to access elements in an array is the FOR...NEXT statement.

DIM STATEMENT - ONE DIMENSIONAL ARRAY

We have discussed array and subscript. However, the computer has to be told whether a variable is simple or an array variable. To declare to the computer that a variable will be used as an array, we must make this declaration with the **DIM** statement. The general form of the DIM statement is:

```
Ln    DIM    varname  ( n )
```

This statement consists of a statement number (Ln), the keyword DIM followed any BASIC variable name, parenthesis and the size of the array with the parentheses. The DIM statement is always placed in the beginning of the program. However, it must come before the statement where the array is used. Most computers specify a maximum size of 10 by default, if not otherwise specified by the programmer. However, it is advisable to use the DIM statement all the time. For example,

```
10   DIM   J (5)
```

In this statement, we have informed the computer to create an array variable J consisting of five memory spaces, one for each element, and to label each memory space from 1 to 5. The number in parentheses is not a subscript but the size of the array J. The memory would look as shown above. However with this statement, in some computer systems, a total number of memory spaces could be six instead of five. These systems always label from 0 to 5, instead of 1 to 5. This default system would not create a problem, unless you refer to an array collectively instead of element by element. Nevertheless, in such cases, there are ways to inform the computer not to start from 0. That is, by adding the following statement before the DIM statement, you can control the size of an array;

```
Ln    OPTION    BASE    0  or  1
```

The statement consists of line a number, the keywords OPTION and BASE followed by either 0 or 1. However, systems programmed not to start with zero would not require this statement.

To rewrite the Program 7.1:

PROGRAM 7.2

```
10  DIM J(5)
20  T = 0
30  V = 0
40  FOR I = 1   TO    5
50  READ J(I)
60  T = T + J(I)
70  NEXT I
80  A1 = T /(I-1)
90  FOR I= 1 TO 5
100  V = V + ( J (I) - A1) ^ 2
110  PRINT J(I)
120  NEXT I
130  V1 = V / ( I - 2 )
140  PRINT "THE AVERAGE OF THE ABOVE NUMBERS IS "; A1
150  PRINT "THE VARIANCE OF THE ABOVE NUMBER IS "; V1
160  DATA 50,70,85,99,65
170  END
```

This program looks longer than Program 7.1, but it is more versatile. You can read more numbers without changing the program except for the DATA statement and the limit value of the FOR...NEXT statements. You can also use the INPUT statement instead of the READ..DATA statements. You can also control the limit value with the INPUT statement.

When line 10 of the program is executed, the computer creates a memory space for five elements and labels those spaces as J and each space as 1, 2, 3, 4, and 5. Line numbers 20 and 30 are used for counters. Each counter is a simple variable. Therefore, the computer creates a single memory space for each variable and labels them with the variable names. Then in line 40, the loop starts with the index I. This loop creates a subscript for the array elements. Line 50, the READ statement, will become READ J(1) when the loop is executed the first time, and will read a value from the DATA statement and put it in J(1) memory. That value is also added to the counter in line 60. The loop continues until five values are read and added to the counter. Line 80 calculates the average and storage in memory space labeled as A1. Then the next loop calculates the differences by subtracting the average from each of the values which are still in the memory of the computer. In line 130, the variance is calculated. The output of this program is as follows:

OUTPUT 7.1

```
50
70
85
99
65
The average of the above numbers is  73.8
The variance of the above numbers is 354.72
```

As we saw in the above program, the FOR...NEXT statements are better for controlling the subscripted variables (array).

Suppose you want to read a set of numbers and then print them in reverse order. For example,

PROGRAM 7.3

```
10  DIM   A (10)
20  FOR  J = 1  TO  10
30  READ  A(J)
40  NEXT J
50  FOR  J =  10  TO  1  STEP  -1
60  PRINT A (J)
70  NEXT J
80  DATA 20,90,60,50,30,70,80,35,40,65
90  END
```

In this program, the array is reversed by changing the index initial and limit values. The output of the program is as follows:

OUTPUT 7.2

```
65
40
35
80
70
30
50
60
90
20
```

The most important use of an array is sequential ordering of numbers. That is, we can sort value in ascending or descending order with the use of an array. However, let us first learn how to find the largest value and the smallest value among the ten numbers. This will require reading all the numbers in to an array and checking for the smallest and the largest values amongst them. The following program will accomplish this:

PROGRAM 7.4

```
 10   DIM  A(10)
 20   FOR  I = 1 TO 10
 30   READ A(I)
 40   PRINT A(I);
 50   NEXT I
 60   PRINT
 70   V1  =  A(1)
 80   V2  =  A(1)
 90   FOR  I  = 2  TO  10
100   IF A(I) >= V2 THEN  V2 = A(I)
110   IF A(I) <= V1 THEN  V1 = A(I)
120   NEXT I
130   PRINT " THE LARGEST VALUE IS "; V2
140   PRINT " THE SMALLEST VALUE IS "; V1
150   DATA 20,90,60,50,30,70,80,35,40,65
160   END
```

We read the ten numbers first. In line 70, we assume that the first element of the array is the smallest and assign it to variable V1. In line 80, we assume that the first element of the array is the largest and assign it to variable V2. Even though this may not be true, but we need a

value to compare with each succeeding element to determine which is the smallest and which is the largest. In line 110, we compare V1 with the each element. If we find an element smaller than V1, we assign the element to V1 and continue until all the elements are checked. By the time the loop is completed, the smallest among the elements will have been assigned to V1. The same reasoning applies to line 100. The output of the program would look as follows:

```
OUTPUT 7.3

20  90 60 50 30 70 80 35 40 65
THE LARGEST VALUE IS  90
THE SMALLEST VALUE IS  20
```

The use of an array is not limited to the examples discussed so far. We can copy one array into another array. For example,

```
PROGRAM 7.5

10  DIM  A(10), X(10)
20  PRINT "X    A"
30  FOR I = 1 TO 10
40  READ A(I)
50  X(I) = A (I)
60  PRINT X(I), A(I)
70  NEXT I
80  DATA 20,90,60,50,30,70,80,35,40,65
90  END
```

In the program, the data is read and stored in the array A, and in line 40 it is also stored in the array X. Keep in mind that both arrays have been declared in line 10.

```
PROGRAM 7.4

X   A
20  20
90  90
60  60
50  50
30  30
70  70
80  80
35  35
40  40
65  65
```

Suppose we want to modify the Grade program so that we print each examination grade and the average grade of each student. In addition, we want to find the average grade of the whole class in each examination and the overall average for the class. The modified program would look as follows:

```
                    PROGRAM 7.6

10 DIM  G(10), Z(3)
20 T1 = 0
30 PRINT " HOW MANY STUDENTS YOU WANT TO ENTER";
40 INPUT M
50 FOR I = 1 TO M
60 PRINT "TYPE THREE EXAMINATION GRADES FOR EACH STUDENT"
70 T = 0
80 FOR K = 1 TO  3
90 INPUT G(K)
```

(continued on next page)

PROGRAM 7.6 (continued)

```
100 T = T + G(K)
110 Z(K) = Z(K) + G(K)
120 NEXT K
130 A2 = T/3
140 PRINT"Examination scores of this student are";G(1);G(2);G(3)
150 PRINT " and the average grades is "; A2
160 T1 = T1 + A2
170 NEXT I
180 A1 = T1 / (I-1)
190 FOR  K  =  1 TO 3
200 Z(K) = Z(K)/3
210 NEXT K
220 PRINT "Average examination scores and overall average"
230 PRINT "for this class are "; Z(1), Z(2), Z(3), A1
240 END
```

In the program, line 10 declares two arrays, G for storing grades and Z for keeping the total for each examination grade. You might think that since there are three examinations, why not have a simple variable to total each examination. Even though that is possible, it is much easier to keep the running total in an array which makes it easier to print in some situations. We still need two counters, T for keeping the total of each student's grades, and T1 for keeping the total of the whole class.

In line 70, T is initialized to zero (0) for each stu dent. That is, the three examinations of each student are totaled in line 100, the average is calculated in line 130, and print in line 150. The T is initialized back to zero to be used for another student. In line 110, array Z is adding up each examination for the whole class. In lines 190 through 210, the average of each examination is calculated and printed in line 230 along with the average for of all the three examinations for the whole class.

The output of the program is:

```
                OUTPUT 7.5

HOW MANY STUDENTS YOU WANT ENTER? 3
TYPE THREE EXAMINATION GRADES FOR EACH STUDENT
? 40
? 60
? 70
Examination scores of this student are 40   60   70
and average grade is 56.6666
TYPE THREE EXAMINATION GRADES FOR EACH STUDENT
? 90
? 99
? 95
Examination scores of this student are 90   99   95
and average grade is 94.6666
TYPE THREE EXAMINATION GRADES FOR EACH STUDENT
? 86
? 88
? 99
Examination scores of this student are 86   88   99
and average grade is 91
Average Examination scores and overall average in this class is
72            82.3333       88            80.7777
```

In this output, three examination grades for each of the three students were entered. The computer printed each examination grade and the average for each student. At the end of the program, the average score in each examination and the overall average score of the class was printed.

DIM STATEMENT - TWO-DIMENSIONAL - Matrix (Table)

Array allows for storing a list of data. Each value is accessed by referencing its subscript. That is, one subscript is required to access value in an array. However, there are situations which require storing more that one variable under one collective name. For example,

TABLE 7.1

ROW	COLUMN 1	COLUMN 2	COLUMN 3	COLUMN 4
		Examinations		
	Student	1	2	3
1	Khalid	90	99	95
2	Riaz	86	88	99
3	Said	40	60	70

The above table has names and grades of three students in rows 1,2, and 3, and the name of the student and his three examination grades in columns 1,2,3, and 4. That is, the rows refer to each student, and columns refer to the examination grades. For example, the data in row one represents the three examination scores of Khalid. Thus, the score of Khalid in the first examination can be found in row 1 and column 2. On the other hand, the score of Said in the third examination can be found in row 3 and column 4.

This arrangement of information in rows and columns is referred as a **two-dimensional table, or a matrix.** In our example, the two-dimensional table is comprised of three rows and four columns, a total of 12 data items (3 x 4). However, we cannot use the same table name for numeric data as well as for character data. Therefore, we will have to declare an array for the names column and a table for the grade columns.

Like an array, a table is identified by a variable name. The naming of a table follows the same syntax rules as a simple variable. However, to identify an element in a table, the table along the row and the column number must be given. That is, an element is referred to by specifying the table name and two subscripts. The subscripts are enclosed in parentheses and separated by comma. The first subscript is the row number and the second subscript is the column number.

The table name must be declared in the DIM statement. The general format of the DIM statement for declaring a table is:

```
Ln    DIM    varname (n of rows, n of columns)
```

This statement consists of a line number (Ln), the keyword DIM, table name followed by parentheses and the number of rows and the number of columns that a table would consists of. For example,

```
10    DIM    G ( 6, 5)
```

This statement declares a table named G with the maximum number of 6 rows and 5 columns. When the computer executes this statement, it will create a memory block as follows:

G

	1	2	3	4	5
1	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)
2	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)
3	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)
4	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)
5	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)
6	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)

The computer creates the above memory and a reference could be made to each cell, the intersection of a row and a column, by providing the two subscripts of the cell as shown above. For example, to access a value in row six and column four, we will specify G (6,4). The left subscript is the row, and the right subscript is the column. As noted previously, the subscript could be a constant, a variable, or an expression. The FOR..NEXT statements are commonly used to specify the subscripts. By varying the row subscript, we move from one row to another. By varying the column subscript, we move from one column to another.

Let us write a program that reads and prints three rows and two columns of numbers:

PROGRAM 7.7

```
 10   DIM  N(3,2)
 20   FOR I = 1  TO 3
 30   FOR J = 1  TO 2
 40   READ N(I,J)
 50   NEXT J
 60   NEXT I
 70   PRINT "          COLUMN"
 80   PRINT "  ROW    1     2"
 90   FOR  I =  1  TO  3
100   PRINT I;
110   FOR J = 1 TO  2
120   PRINT N (J,I);
130   NEXT J
140   PRINT
150   NEXT I
160   DATA 90,99,95,86,88,99
170   END
```

The output of the program is as follows:

```
OUTPUT 7.6

        COLUMN
ROW      1     2
 1      90    99
 2      95    86
 3      88    99
```

The computer initializes index I, line 20, and then index J, line 30. It reads the value 90 in the data statement and stores it in row one and column one, of table N. The inner loop is executed twice. It means the next value, 99, is stored in N(1,2). After executing the inner loop, the value of index I is increased by 1. The next values to be read will be 95 and 86 and will be stored in cells N(2,1), N(2,2) respectively. The index I is increased by 1. The next values to be read will be 88 and 99 and will be stored in cells N(3,1), N(3,2).

Let us write a program to load the data in Table 7.1 into the computer memory.

```
PROGRAM 7.8

 10   DIM  N$(3), G(3,3)
 20   FOR  I = 1  TO  3
 30   READ N$ (I)
 40   FOR  J = 1  TO  3
 50   READ G(I,J)
 60   NEXT J
 70   NEXT I
 80   DATA Khalid,90,99,95,Riaz,86,88,99
 90   DATA Said,40,60,70
100   END
```

In this program, line 10 declares one array, N$, of size 3 for storing the names, and table G of sizes 3 by 3, for storing three examination scores each for the three students. In line 20, index I is initialized to one. In line 30, the READ statement reads the name of the first student in the DATA statement and stores it in N$(1), and then the index J is initialized to 1. The READ statement reads the examination score of the first student and stores in the table. The memory, if we could see, would look as follows:

COMPUTER MEMORY

	N$ 1	G 1	G 2	G 3	I	J
1	Khalid	90			1	1
2						
3						

Since the computer completes the inner loop first, the J will be changed to 2. The READ statement in line 50 will read the second examination score of the same student and store it in the computer memory in row one and column two of Table G. Then the third examination score will be read and stored in the first row and the third column of Table G. Since the J exceed the limit value, the outer loop will be executed, the value of I will be changed to 2. The READ statement, line 30, will read the name of the second student. The index J will be initialized to 1. The first examination score of the second student will be read and stored in the computer memory. The computer memory would look as follows:

COMPUTER MEMORY WHEN I = 2 and J = 1

	N$ 1	G 1	G 2	G 3	I	J
1	Khalid	90	99	95	2	1
2	Riaz	86				
3						

This process will continue until the names of the students and their grades have been read. The memory of the computer would look as follows:

COMPUTER MEMORY WHEN I = 3 AND J = 3

	N$	G				
	1	1	2	3	I	J
1	Khalid	90	99	95	3	3
2	Riaz	86	88	99		
3	Said	40	60	70		

Let us rewrite the Grade program. The new grade program using the two-dimension format would look as follows:

PROGRAM 7.9

```
 10 T1 = 0
 20 DIM N$(10),G(10,3),Z(3)
 30 PRINT"HOW MANY STUDENTS YOU WANT TO  GRADE";
 40 INPUT M
 50 PRINT
 60 PRINT
 70 PRINT"                  GRADE   SUMMARY"
 80 PRINT
 90 PRINT" NAME             EXAMINATION GRADE       AVERAGE"
100 PRINT"-------------------------------------------------------"
110 FOR I = 1 TO M
120 READ N$(I)
130 PRINT USING " \          \ \     \";N$(I);"   | ";
140 T = 0
150 FOR K = 1 TO 3
```

(continued on next page)

PROGRAM 7.9 . (continued)

```
160 READ G(I,K)
170 T = T + G(I,K)
180 Z (K) = Z(K) + G(I,K)
190 PRINT USING " ###.##  \  \"; G(I,K);"    | ";
200 NEXT K
210 A2 = T/3
220 PRINT USING " ###.##";A2
230 T1 = T1 + A2
240 NEXT I
250 A1 = T1 / (I-1)
260 FOR I = 1 TO 3
270 Z(I) = Z(I) / M
280 NEXT I
290 PRINT"----------------------------------------------------------"
300 PRINT" AVERAGE        |      ";
310 PRINT USING" ###.## \   \  ";Z(1);" |";Z(2);" |";Z(3);" |";A1
320 DATA Khalid,90,99,95,Riaz,86,88,99,Said,40,60,70
330 END
```

In the program the names are read in line 160 and stored in array N$; then the grades are read in line 160 and stored in G(I,K) table. The grade of each student is added in line 170, T.

Line 180 adds each examination grades. The grades are printed by the format in line 190. The rest of the statements are the same in the previous Grade program. The output of the program is as follows:

OUTPUT 7.7

HOW MANY STUDENTS YOU WANT TO GRADE? 3

GRADE SUMMARY

NAME	EXAMINATION GRADE			AVERAGE
Khalid	90.00	99.00	95.00	94.67
Riaz	86.00	88.00	99.00	91.00
Said	40.00	60.00	70.00	56.67
Averages	72.00	82.33	88.00	80.78

In this output, besides the titles, each student's grades are printed and the average of the three grades is printed at the end of the row. The average of each examination is printed along with the average for the class.

You can modify this program to read any number of students by inputting larger numbers for the number of students to be graded. You can change the number of examinations you want to read by changing the number in the FOR statement in line 150 from 3 to any number of examinations you want to read. Keep in mind that you must provide the appropriate number of students and examinations in the DATA statement. If you want to input the data every time the program is run, and is necessary to change the READ statements to INPUT statement in lines 120 and 160.

QUESTIONS

1. Define an array and a matrix.

2. What does the DIM statement do?

3. What would happen to an array if the DIM statement was not specified in the program?

4. What is a subscript and why it is needed?

5. Write a program that reads and prints grades, one for each student, of ten students and calculates the average grade for the class. Use an array.

6. Write a program that calculates average and variance of the numbers between 10 and 20 inclusive. You must use a counter and must not read or input the numbers into the program.

7. Write a program that reads the height and the weight of a man and determines whether he is overweight, under-weight, or average weight. Use Do loop to generate the following table values and store them in an array.

Height	Average Weight
5' - 1"	128 - 134
5' - 2"	134 - 140
5' - 3"	140 - 144
5' - 4"	143 - 149
5' - 5"	147 - 153
5' - 6"	152 - 158
5' - 7"	156 - 163
5' - 8"	161 - 167
5' - 9"	166 - 172
5' - 10"	171 - 176
5' - 11"	175 - 181
6' - 0"	181 - 186
6' - 1"	186 - 191
6' - 2"	191 - 196
6' - 3"	197 - 200

8. Write a program that reads the wages of ten employees and deducts income tax based on the following table. Use the Do loop to create the table. Also, your program should calculate the average and the variance of the wages.

	Wages	Tax Rate
Rs.	0 - 10000	0 %
	10001 - 20000	2 %
	20001 - 30000	4 %
	30001 - 40000	6 %
	40001 - 50000	8 %
	60001 - 70000	10 %
	80000 - 90000	12 %
	100000 and over	14 %

9. Show the number of lines and the values that will be printed by each of the following program segments.

a)
```
FOR I = 1 TO 5
Y(I) = I
NEXT I
PRINT Y;
```

b)
```
DIM W (10)
FOR I = 1 TO 20 STEP 5
W(I) = I
NEXT I
PRINT W(I), W(I+1), W(I+2), W(I+3)
```

c)
```
DIM W(20), Y(20)
FOR I = 1 TO 100 STEP 10
W(I) = I
Y(I) = I + 1
PRINT I
NEXT I
PRINT W, Y
```

d)
```
FOR J = 1 TO 2
FOR K = 1 TO 3
W(J) = J
Y(K) = K
PRINT J,K
NEXT K
NEXT J
```

10. Correct errors, if any, in the following program segments:

a)
```
FOR  I = 10 TO 30  STEP -1
Y(I) = I
NEXT I
```

b)
```
FOR J = 1 TO 10
J = Y(J)
NEXT I
```

d)
```
FOR I = 40 TO 0 STEP 2
Y(K) = I
NEXT I
```

e)
```
FOR J = 1 TO 15
W(J) = J + 1
PRINT J
NEXT J
```

c)
```
FOR L = 1 TO 10
FOR K = 1 TO 2
FOR J = 1 TO 3
W(J) = J
X(K, J) = K
Y(L, K) = K
PRINT W(J), X (K, J), Y (L,K)
NEXT J
NEXT L
NEXT K
```

11. Write a program that prints two columns of numbers. The first column should have the numbers between 10 and 20 and the second column between 40 and 60. You should not input or read data.

12. Write a program that will read the following table of numbers and will print the average and the variance of each row and each column.

10	20	44	45	50	80	90	50
60	30	66	43	25	29	40	60
40	50	60	40	30	20	34	43
35	99	20	39	50	60	77	88

13. Write a program that reads 15 numbers and then sorts and prints them first in ascending and then in descending order.

14. Suppose array A has been created as follows:

Location	1	2	3	4	5	6	7	8	9	10
Value	5	10	6	9	12	4	8	13	3	11

Assume that the value of L is 5 and M is 4. What value will each of the following statements print?

a) PRINT A(L)
b) PRINT A(M)
c) PRINT A(M+L)
d) PRINT A(L+1)
e) PRINT A(M+1)

15. Suppose matrix A has been created as follows:

Location	1	2	3	4	5	6	7	8	9	10
	5	10	6	9	12	4	8	13	3	11
	6	11	7	10	13	5	9	14	4	12
	7	12	8	11	14	6	10	15	5	13
	8	13	9	12	15	7	11	16	6	14

Assume that the value of L is 5 and M is 4. What value will each of the following statements print?

a) PRINT A(L,2)
b) PRINT A(3,M)
c) PRINT A(4,M+L)
d) PRINT A(3,L+1)
e) PRINT A(2,M+1)

16. A company sells four products with five models of each product. The following table gives the prices of each model of each product.

Price of each Product and Model

Product		Model 1	2	3	4	5
1	Rs.	5	10	6	9	12
2		6	11	7	10	13
3		7	12	8	11	14
4		8	13	9	12	15

Write a program that prints the customer number, the quantity sold, the price of each product and model, and the sales amount next to the customer number and the quantity sold to each customer. All of this information should be printed in a tabular form.

17. Write a program that adds, subtracts, and multiplies two arrays.

18. Write a program that adds, subtracts, and multiplies two matrices.

19. Write a program that reads survey results of the income distribution in different cities and summarizes the results as follows.

Income Distribution by Cities

Number of People in each City

Income Level	Peshawar	Lahore	Karachi	Multhan	Sind
0 - 10000					
10001 - 20000					
20001 - 30000					
30001 - 50000					
50001 and over					

CHAPTER 8

SUBPROGRAMS - FUNCTIONS AND SUBROUTINES

FUNCTIONS:

A function is a predefined algorithm. That is, it performs some calculations. Some functions return values or a value to the statement which makes the reference (call) to the function each time within the program.

BASIC provides two kinds of functions:

1. **Built-in functions** are predefined within the language.

   The built-in functions can be classified as:

   a) Mathematical
   b) System
   c) String
   d) Matrix

   However, only the mathematical, string, and matrix functions will be discussed in this book. The matrix function will be discussed as an appendix, and the other two will be discussed in this chapter.

2. **User-defined functions** are defined by the programmer.

   There are two types of user-defined functions:

   a) Single-line
   b) Multiple-line

Built-in Functions:

Mathematical Function:

There many mathematical operations that we need for solving business or mathematical problems. Luckily, most of these operations are already programmed by the computer vendors. We just have to find the correct name of the programmed operation (function) and use it as a part of our program. The general form of the function statement is:

```
Ln    varname = function name (value or expression).
```

The statement consists of the line number (Ln), a variable name, the equal sign, the name of the function, and a value or an expression commonly referred to as the function argument. The value or argument is passed on to the function to work on. For example, to find the square root of a number, we need to tell the computer the name of the function and the value of which we want to find the square. That is,

```
10   T  = SQR(9).
```

In this statement, the function name is SQR. By specifying the SQR, we have asked the computer to find the program named SQR, calculate the square root of the number nine, and store the result in the variable named T. These programs are referred as a built-in functions. This is one of the mathematical functions available on most BASICs. Table 8.1, lists the most common numeric functions:

TABLE 8.1

Numeric Functions

Function	Description
ABS (Y)	Finds the absolute value of Y. For example, `10 V = ABS (-10)` RESULT: V = 10
ATN (Y)	Finds the angle in radians, the arctangent of Y. For example, `10 T = ATN(10)` RESULT: T = 1.471128
COS (Y)	Finds the consine of the Y. Y is in radians. For example, `10 T = COS(10)` RESULT : T = -.8390716
EXP (Y)	Finds e (2.71828) raised to the power of Y. For example: 10 T = e^{10} RESULT: T = 22026.32
FIX (Y)	Finds the integer (whole number) value of the a fraction. For example, `10 T = FIX(67.323) or FIX(67.67)` RESULT : T = 67

INT (Y) Finds the truncated number (integer) of the positive value of Y. For example:

```
10    T  = INT (6.7).
```

RESULT :   T = 6.

OR

Finds the greatest integer that is less than or equal to Y. For example,

```
10  T  = INT (-6.3)
```

RESULT : T = -7.

In this case, the -7 is the greatest integer less than -6.3.

LOG(Y) Finds the natural logarithm for Y > 0. It is the reverse of the EXP(Y) function. For example,

```
10  T = LOG(20)
```

RESULT : T = 2.995732

RND Finds the random numbers between 0 and 1. It could also be written as RND (Y), but most computer do not require the value or argument Y.

SGN(Y) Finds the sign of Y. There are three possible values : -1, 0, 1. If Y is positive non-zero number, the return value will be 1; if Y is zero, the returned value will be 0; if Y is negative, the returned value will be 1. For example,

```
10  T =SGN(10)
```

RESULT :   T = 1

SQR(Y) Finds the square root of a positive number Y. For example,

```
10 T = SQR(10)
```

RESULT : T = 3.162278

NOTE: Some BASICs may not have some of the above functions. Check your system's book for available functions.

Let us write a program using all these built-in functions.

PROGRAM 8.1

```
10  Y = 10
20  T =  EXP (Y)
30  PRINT " EXP. FUNC. "; T, EXP(X)
40  T = ATN(Y)
50  PRINT "ARCTANGENT "; T, ATN(Y)
60  T = COS(Y)
70  PRINT " COSINE "; T
80  T = SGN(Y)
90  PRINT "SGN  " ; T
100 T = SQR(Y)
110 PRINT "SQUARE ROOT",T
120 Y = -10
130 T = INT(Y)
140 PRINT " INTEGER VALUE "; T
150 T = ABS(Y)
160 PRINT "ABSOLUTE VALUE "; T
170 T = LOG(20)
180 PRINT "LOG  "; T
190 PRINT " RANDOM NUMBERS"
200 FOR I= 1 TO 10
210 T = RND
220 PRINT T
230 NEXT I
240 END
```

In some print statements, you will find the variable name and function name in the same statement. The purpose of this repetition is to show you that the results of a function could be printed without first being stored in a variable. The output of the above program is as follows:

```
                  OUTPUT 8.1


EXP. FUNC.     22026.47              22026.47
ARCTANGENT     1.471128              1.471128
COSINE        -.8390716
SGN             1
SQUARE ROOT   3.162278
INTEGER NUMBER    -7
ABSOLUTE VALUE     6.3
LOG           2.995732
RANDOM NUMBERS
 .7151002
 .683111
 .4821425
 .9992938
 .6465093
 .1322918
 .3692191
 .5873315
 .1345934
 .9348853
```

Random Numbers:

In the above output, we generated ten random numbers. The function RND was used for this purpose. The RND function is very useful for programmers who are involved in simulation. Through a repeatable process, the RND function generates **pseudo-random numbers**. Each time the function is used, any decimal fractional number between 0 and less than 1 has an equal chance of being selected.

However, any time the same program with an RND function is executed, it will generate the same set of numbers. If a different set is needed, the following statement must precede the RND function:

```
Ln  RANDOMIZE
```

For example, the following set of statements will generate a different set of numbers every time they are executed.

```
PROGRAM 8.2

10  RANDOMIZE
20  FOR  I = 1 TO 10
30  PRINT RND
40  NEXT I
50  END
```

The outputs of the above program run twice are:

```
OUTPUT 8.2

RANDOM NUMBER SEED (-32768 TO 32767)? 324500

 .6646266
 .2031866
 .9959512
 .2943542
 .3104612
 .4695987
 .4284047
 .2433319
 .7245145
 .4796363
```

(continued on next page)

OUTPUT 8.2 (continued)

```
RANDOM NUMBER SEED (-32768 TO 32767)? 30300
 .7623666
 .4401157
 .58503
 .5689271
 .3027215
 .0448879
 .01121293
 .5455051
 .6193254
 .3443367
```

String Functions:

Many applications require the manipulation of characters in a data. That is, some time we may want to modify, concatenate (add together), compare, and analyze a set of strings. BASIC provides many string functions to perform these operations. The general form of the string function is the same as mathematical functions, except the dollar sign ($) is added at the end of the function name. For example,

```
Ln   var = function name$(var or value)
```

Substring

A substring is a group of one or more adjacent characters in a string. For example, in string **Shahabuddin**, the

substrings are

S
Shah
abudd
in
Shahab

All these are considered substrings. That is, any adjacent characters like **Shah or abudd or in or Shahab** in a string are substrings.

BASIC provides functions to extract substrings from a string. The most common functions are listed in Table 8.2.

TABLE 8.2

Substring Functions

Function	Description
LEFT$(var or str,N)	Returns the first N characters of the variable or string. For example, A$ = LEFT$("Shahabuddin",5) RESULT : A$ = Shaha
RIGHT$(variable,N)	Returns the rightmost N characters of the variable or string. For example, A$ = RIGHT$("Shahabuddin",5) RESULT : A$ = uddin
MID$(var or str,N,M)	Returns M number of characters starting at N. For example, A$ = MID$(Shahabuddin,4,3) RESULT : A$ = hab

In this example, three characters "hab" starting at location 4 are extracted.

A$(M:N) Returns M through N character of string variable A$.

LTRM$(var or str) Removes all leading spaces in a string variable or a string. For example,

```
A$ = LTRM$("   Shahabuddin")

RESULT :    A$ =Shahabuddin
```

In this example, the blank spaces before the string "   Shahabuddin" are removed.

RTRM$(var or str) Removes all trailing spaces in a string variable or string. For example,

```
A$ = RTRM$("Shahabuddin      ")

RESULT :    A$ =Shahabuddin
```

In this example, the blank spaces after the name are removed.

LPAD$(var or str,N) Increases (pads) the size of the string by adding spaces on the left of the string. For example,

```
A$ = LPAD$("Shah",6)

RESULT :  A$ =   Shah
```

In this example, two spaces were added to the left of the string.

RPAD$(var or str,N) Increases the size of the string by adding spaces on the right of the string. For example,

```
A$ = RPAD$("Shah",6)

RESULT :  A$ =Shah
```

In this example, two spaces were added to the right of the string.

RPT$(var or str,K)	Creates a string by repeating the characters in the string K number of times. For example, A$ = ("Shah",3) RESULT : A$ = ShahShahShah

NOTE: Some BASICs may not have all the functions. Check the your systems' BASIC book for available functions.

In addition to the truncation functions, Table 8.4 lists additional string functions.

TABLE 8.4

OTHER STRING FUNCTIONS

Function	Description
LEN(var or str)	Determines the length (the number of characters) of the variable or string. For example, M = LEN("SMALL") RESULT : M = 5
ASC(var or str) ASCII(var or str) or ORD(var or str)	Finds the ASCII code for the character of string. For example, B = ASC("A") or ASCII("A") RESULT : B = 65

CHR$(var or str)	Finds the character string or variable of the ASCII code. For example,

```
A$ = CHR$(66)

RESULT :  A$ =  B
```

VAL(var or str)	Finds the numeric equivalent in the string or variable. For example,

```
B = VAL("A")

RESULT :  B = 0
```

In this example, there is no number in the "A".

```
B = VAL("1306 Preston")

RESULT :  B = 1306
```

STR$(var or number)	Finds a string equivalent of a number or numeric variable. For example,

```
B$ = STR$(10)

RESULT :  B$  = 10
```

POS(var or str,str1,K)	Finds the position of str1 within a string or variable. The position search begins at position K. For example,

```
A = POS("Shahabuddin","b",2)

RESULT :  A = 6
```

In this example, the string "shahabuddin" is searched for character "b" starting at position 2. It finds the character "b" at position 6th in the string.

Keep in mind that in most microcomputers, the POS function returns the current cursor position on the screen.

UPRC$(var or str)	Replaces all lower-case characters with their uppercase equivalents. For example A$=UPRC$("Shahabuddin") RESULT : A$ = SHAHABUDDIN
LWRC$(var or str)	Replaces all uppercase characters with their lower-case equivalents. For example, A$=LWRC$("SHAHABUDDIN") RESULT : A$ = shahabuddin

---

NOTE: Check your system's BASIC for available functions.

USER-DEFINED FUNCTIONS:

As discussed in the preceding section, many functions are available in BASIC. However, there are many situations where the available built-in functions may not meet the needs of the programmer. These situations may require writing our own functions. For such purpose, BASIC allows programmers to write their own functions, called **user-defined** functions. These are written as one-line statements as a part of the program and referenced (called) from anywhere in the program. The general format of a user defined function is:

```
Ln  DEF  FNletter(dummy argument) = expression.
```

In this statement, the line number (Ln) is followed by the keywords DEF and FN. The DEF is used by the BASIC statement for recognizing the user-defined function in the program. The FN keyword should be followed by a letter A through Z. The dummy argument (parameter) within the parentheses could be any numeric or string BASIC variable name followed by an expression to the right of the equal sign.

Expression is any mathematical expression which may contain a built-in function or other user-defined function. The DEF statement is a non-executable statement and must precede the statement where it is used. It can appear alone or as a part of another function. For example,

```
10    DEF FNZ(X) = SQR(9) * X + X
```

In this statement, the FNZ function will use the variable X for calculating the expression on the right of the equal sign. The expression on the right calculates the square root of 9 multiplied by the variable X and added to variable X.

Once a function is defined, it can be referenced at any time, anywhere in the program by using its three-letter function name and a value or a defined variable. For example, the following statement in the same program could call the function FNZ defined in the statement number 10 above.

```
120   LET R1 = FNZ(10)
```

When the computer executes this statement, it starts looking (calls the DEF function) for a function FNZ in the program. It will find statement 10 in the program and make the value of X equal to 10. It will calculate the expression on the right of the equality and return to statement 120 and make the value of R1 equal to the result of statement 10. That is, the value of R1 will be 40. Statement 10 can be used as many times in the program as one wants. For example, another statement in the same program could call the same function using a different value:

```
150   LET G1 = FNZ(36)
```

This statement 150 also calls statement 10. This time the value of X will be 36. The expression in statement 10 will be evaluated and the results returned to statement 150 and stored in variable G1.

The dummy variable (argument) assigned to the function as a parameter is local to the function. That is, a memory space is not created for it until the function is executed. For example,

```
 20   DEF FNR(R,P,Y) = P * (1+R)^Y

          ....

          ....

160   READ R,P,Y
170   A = FNR(R,P,Y)
```

In this partial program, the function in line 20 is calculating compound interest. When R, P, and Y are read in line 160, the dummy variables R, P, and Y in line 20 are not affected (not defined) . That is, they are considered distinct from any variables with the same name outside of the function definition.

However, variables in the function's expression could be defined outside the function even though they are not listed as dummy variables in the function. For example, to rewrite the above program:

```
20 DEF FNR(R) = P * (1+R)^Y

      ....

      ....

160 READ R,P,Y
170 A = FNR(R)
```

In this partial program, the variables P, R, and Y in the function statement are the same as in the READ statement. When the READ statement is executed, the variables R, P, and Y are assigned values. However, since no call has been made to the function yet, the R variable in the function is not assigned value yet. The other two variables, P and Y in the function expression will be defined when the READ statement is executed.

The functions can be called without writing them as a part of an assignment statement, i.e. as statement 170 below. For example, to rewrite the above statements:

```
 20 DEF FNR(R) = P * (1+R)^Y

       ....

       ....

160 READ P,Y
170 PRINT  FNR(1),FNR(2),FNR(3)
```

In this partial program, only values for P and Y are read. In line 170, the PRINT statement calls the function three times and three different values of R are passed on. That is, the PRINT statement makes a call to calculate amounts for one year, two years, and three years. The program will print three amounts.

As shown so far, the purpose of a use-defined function is that the same expression may be needed in many statements of the program. Therefore, instead of repeating the same expression in every statement of the program, it is better to write it as function and then make a call to it.

SUBROUTINE

The user-defined functions perform important functions. However, they are limited to one statement. In some situations the programmers may want to access a set of statements over and over again in the program. This can be placed in a **subroutine** which is a set of statements that performs a particular function and may be needed in several parts of the program. Subroutines are usually placed at the end of a program which is called a **main program**.

The use of subroutines is becoming more common due to the popularity of structured programming and the use of modular programming. Since subroutines are designed to perform certain activities, one can develop a separate module (subroutine) for each activity and test and debug it before making it a part of the main application program. Subroutines are small and related to one activity. Therefore, they increase the efficiency of programmers and make debugging easier. The main program acts as a controller by making calls and receiving messages from the subroutines.

A subroutine can begin with any BASIC statement and ends with a **RETURN** statement. The general form of a subroutine is as follows:

```
Ln   any BASIC statement

 .  |
 .  |
 .  |>- set of BASIC statement
 .  |
 .  |

Ln  RETURN
```

In a subroutine, the first statement could be any BASIC statement. For clarity, it is advisable that this statement should be a REM statement to identify the subroutine. For example,

```
300  REM  SUBROUTINE  INTEREST
```

This statement is followed by any number of other BASIC statements needed to solve a particular problem. The last statement must be the RETURN statement (which is used only in a subroutine and has no other purpose). When the RETURN is executed, the computer returns to the statement following the statement in the main program where the subroutine was called from and continues the execution of the main program. A program could have as many subroutines as needed provided they follow the above format.

All the subroutines are normally placed at the end of the main program and before the END statement. However, if there is no break between the main program and the subroutines, the computer will continue to execute the subroutine in a normal sequence as a program. However, subroutines are called by the main program and cannot be executed as separate programs. Therefore, there should be a statement after the main program that breaks the normal sequence of the program. (For this purpose use either the GOTO n or the STOP.) However, in some systems, the use of the STOP statement could cause an unwanted message. In that case, use the GOTO n where the GOTO goes to the END statement in the program. Make sure that you choose either the STOP or the GOTO n but not both.

The **GOSUB** is used in the main program for calling a subroutine from the main program. The general format of the call statement is:

```
Ln   GOSUB  Ln1
```

In this statement, the line number (Ln) is the line number of the GOSUB statement followed by the keyword GOSUB and the line number of the first statement of the subroutine. For example,

```
 10  REM  MAIN PROGRAM
 20  READ  P,R,N
 30  GOSUB 300
 40  DATA 1000,.05,10
 50  READ  P,R,N
 60  GOSUB 300
 70  DATA 2000,.05,10
 80  PRINT "DONE "
              .
              .
              .
290 STOP (or) GOTO 350    Note : choose either STOP or GOTO 350
300 REM SUBROUTINE INTEREST CALCULATION
310  A = P* (1-R) ^N
320  PRINT " AMOUNT  ";A; " PRINCIPAL  ";P
330  PRINT " INTEREST RATE "; R; " YEARS "; N
340  RETURN
350  END
```

This partial program shows that line number 20, the READ statement in the main program reads data for the three variables, P,R, and N. Line number 30 makes a call to the subroutine in line 300. The subroutine calculates the compound amount and prints the amount, principal, interest rate, and the number of years. Line number 340 in the subroutine returns the computer back to the next executable statement following the GOSUB statement. That is, in our program the computer returns to line number 50 in the main program. The next READ statement reads new data and line number 60 makes another call to the subroutine. The subroutine calculates and prints the results and returns back to line number 80. After executing other statements between lines 80 to 280, the computer executes line 290. This statement either ends the program if the STOP command is used or transfers it to line 350 if GOTO 350 is used.

A GOSUB statement can be used within a subroutine to call another subroutine; they are considered nested subroutines. However, there is a limit to the number of nested subroutines. Some BASICs would not allow too many subroutines. Two or three levels are sufficient for most applications. (If you use more than three levels, check your algorithms in order to reduce the number of subroutines.) To write a nested subroutine, let us rewrite the above partial program:

```
 10  REM  MAIN PROGRAM
 20  READ  P,R,N
 30  GOSUB 300
 40  DATA 1000,.05,10
 50  READ  P,R,N
 60  GOSUB 300
 70  DATA 2000,.05,10
 80  PRINT "DONE "
               .
               .
               .
290  STOP (or) GOTO 380  Note : choose either STOP or GOTO 380
300 REM SUBROUTINE INTEREST CALCULATION
310  A = P* (1-R) ^N
320  GOSUB 340
330  RETURN
340 REM SUBROUTINE PRINTING OUTPUT
350  PRINT " AMOUNT  ";A; " PRINCIPAL  ";P
360  PRINT " INTEREST RATE "; R; " YEARS "; N
370  RETURN
380  END
```

In this partial program, in line 320, the CALCULATION subroutine calls the PRINTING subroutine within the subroutine. The control passes to line 340, and results are printed. The RETURN, in line 370, passes control back to line 330. Line 330, the RETURN statement, returns the control back to the main program.

The following program show a complete program with subroutines:

PROGRAM 8.3

```
 10 REM  *********  MAIN PROGRAM *****************
 20 DIM X(500),F(50),M(50),B(50),C(50)
 40 REM C() IS FOR CUMULATIVE FREQUENCIES
 50 REM M() IS MIDPOINTS OF CLASS INTERVALS
 60 REM R() IS FOR CLASS LIMITS
 70 PRINT
 80 LET B1$="PLEASE ANSWER YES OR NO"
 90 PRINT "DO YOU WANT INSTRUCTIONS";
100 INPUT Z$
110 IF Z$="NO" THEN 160
120 IF Z$="YES" THEN 150
130 PRINT B1$
140 GOTO 100
150 GOSUB 730
160 C1=0
170 PRINT
180 PRINT " HAVE YOU TYPED THE DATA IN THE DATA STATEMENT ";
190 INPUT A$
200 IF A$="YES" THEN 260
210 IF A$="NO" THEN PRINT "DO YOU WANT TO INPUT THE DATA"
220 INPUT Z$
230 IF Z="NO" THE GOSUB 730
240 PRINT
250 PRINT "ENTER THE NUMBER OF DATA POINTS ";
260 IF Z$="NO" THEN READ N    ELSE INPUT N
270 IF Z="YES" THEN PRINT "ENTER ONE NUMBER AT A TIME"
280 FOR I=1 TO N
290 IF Z$="NO" THEN READ X(I)   ELSE INPUT X(I)
300 LET C1=C1+1
310 NEXT I
320 PRINT
330 PRINT "DO YOU WANT TO SEE THE DATA IN SEQUENTIAL ORDER";
```

(continued)

(program continued)

```
340 INPUT B$
350 IF B$="NO" THEN 470
360 IF B$="YES" THEN 390
370 PRINT B1$
380 GOTO 340
390 PRINT
400 PRINT "YOUR DATA IN SEQUENTIAL ORDER IS"
410 PRINT
420 GOSUB 1910
430 REM  **  TO PRINT ORDERED DATA
440 FOR I = 1 TO N
450 PRINT X(I);
460 NEXT I
470 IF B$="NO" THEN GOSUB 1920
480 GOSUB 960
490 PRINT
500 PRINT "SMALLEST VALUE IS ";M1
510 PRINT "LARGEST VALUE IS  ";M2
520 PRINT "RANGE OF THE DATA IS ";M2-M1
530 PRINT
540 PRINT "DO YOU WANT FREQUENCY TABLE ";
550 INPUT Y$
560 IF Y$="YES" THEN GOSUB 1200
570 IF Y$="NO" THEN 2500
580 GOSUB 1470
590 REM *** FREQUENCY HISTOGRAM
600 PRINT
610 PRINT
620 PRINT "DO YOU WANT A FREQUENCY HISTOGRAM";
630 INPUT D$
640 IF D$="NO" THEN 2500
650 IF D$="YES" THEN GOSUB 1700
660 GOTO 2500
670 REM
680 REM *********    END OF MAIN PROGRAM *************
```

(continued)

(program continued)

```
690 REM
700 REM *********          SUBROUTINES     *************
710 REM
720 REM ........    SUBROUTINE : INSTRUCTIONS .........
730 REM
740 PRINT "THIS PROGRAM CREATES A FREQUENCY TABLE AND GRAPHS"
750 PRINT "FREQUENCIES OF DATA. YOU CAN ENTER DATA EITHER "
760 PRINT "THROUGH AN INPUT STATEMENT OR TYPE IT IN THE DATA "
770 PRINT "STATEMENT STARTING AT LINE 2460."
780 PRINT " "
790 PRINT "INSTRUCTIONS : WHEN REQUESTED, ENTER THE FOLLOWING"
800 PRINT "                    INFORMATION"
810 PRINT "  1)  THE NUMBER OF DATA POINTS (OBSERVATIONS)"
820 PRINT "  2)  ANSWER 'YES' IF THE DATA HAS ALREADY BEEN"
830 PRINT "      TYPED IN THE DATA STATEMENT. ANSWER 'NO', IF"
840 PRINT "      THE DATA IS NOT TYPED."
850 PRINT "  3)  THE DATA MAY BE ENTERED IN ANY ORDER INTO"
860 PRINT "      THE DATA STATEMENT. HOWEVER, IF YOU WISH"
870 PRINT "      TO SEE THE DATA ORDERED IN DESCENDING ORDER,"
880 PRINT "      TYPE 'YES' WHEN PROMPTED."
890 PRINT "IF THE NUMBER OF OBSERVATIONS (DATA) IS GREATER"
900 PRINT"THAN 500, THE NUMBER OF CLASS INTERVALS EXCEEDS 50,"
910 PRINT "THEN CHANGE THE DIM STATEMENT IN LINE 20 TO"
920 PRINT "THE SIZE YOU WANT."
930 RETURN
940 REM ........  END OF SUBROUTINE INSTRUCTIONS ............
950 REM
960 REM ........    SUBROUTINE SMALL AND LARGE ............
970 REM TO DETERMINE SMALLEST AND LARGEST VALUES
980 LET M1=X(1)
990 LET M2=X(N)
1000 REM DETERMINE THE VALUE
1010 IF X(1)>= THEN 1050
1020 IF ABS(10*X(1))<X(N) THEN 1050
1030 LET X9=X(1)
1040 GOTO 1070
1050 LET X9=X(N)
1060 IF C1<N THEN 1100
```

(continued)

(program continued)

```
1070 LET L=L-3
1080 LET N9=INT(72/L)
1090 LET N8=INT(N/N9)+1
1100 FOR I=1 TO N8
1110 FOR J=1 TO N9
1120 IF N9*(I-1)+J>N THEN 1140
1130 NEXT J
1140 PRINT
1150 NEXT I
1160 PRINT
1170 RETURN
1180 REM ........  END OF SUBROUTINE SMALL AND LARGE ..........
1190 REM
1200 REM ........    SUBROUTINE FREQUENCY TABLE.........
1210 REM
1220 PRINT " ENTER THE NUMBER OF FREQUENCY INTERVALS."
1230 PRINT " IT COULD BE A NUMBER BETWEEN 5 AND 10"
1240 INPUT N1
1250 D = (M2-M1)/N1
1260 B(1) = M1
1270 REM TO SET UP LIMITS
1280 FOR I = 2 TO N1+1
1290 LET B(I) = B(I-1)+D
1300 NEXT I
1310 REM TO DETERMINE CLASS FREQUENCIES
1320 FOR I = 1 TO N
1330 FOR J = 2 TO M1 + 1
1340 IF X(I)>B(J) THEN 1350
1350 LET F(J-1)=F(J-1)+1
1360 GOTO 1380
1370 NEXT J
1380 NEXT I
1390 REM DETERMINES CLASS MIDPOINTS AND CUMULATIVE FREQUENCIES
1400 FOR I = 1 TO N1
1410 LET M(I) = B(I)+B(I+1))/2
1420 LET C(I) = C(I-1) + F(I)
```

(continued)

(program continued)

```
1430 NEXT I
1440 RETURN
1450 REM ........  END OF SUBROUTINE                  .............
1460 REM
1470 REM ........    SUBROUTINE : PRINTING        ...............
1480 PRINT
1490 PRINT
1500 REM FREQEUNCY TABLE
1510 PRINT TAB(26);"FREQUENCY DISTRIBUTION"
1520 PRINT
1530 PRINT TAB(43):"CUM";TAB(52);"REL";TAB(64);"REL CUM"
1540 PRINT TAB(5);"CLASS LIMITS";
1550 PRINT TAB(22);"MIDPOINTS";TAB(34);"FREQ";TAB(43);
1560 PRINT "FREQ";TAB(52); "FREQ";TAB(64); "FREQ";
1570 PRINT "-------------------------------------------------------"
1580 PRINT "---------------------"
1590 FOR I = 1 TO N1
1600 PRINT B(I);TAB(10);" -  ";B(I+1);
1610 PRINT TAB(24);M(I);TAB(34);F(I);TAB(43);C(I);
1620 PRINT TAB(50);
1630 PRINT USING"##.##       ###.###";F(I)/N,C(I)/N
1640 NEXT I
1650 PRINT"--------------------------------------------------------"
1660 PRINT "--------------------"
1670 PRINT "TOTALS";TAB(34);C(N1);TAB(50);C(N1)/N
1670 RETURN
1680 REM ........  END OF SUBROUTINE                  .............
1690 REM
1700 REM ........    SUBROUTINE FREQUENCY       ............
1710 GOSUB 2330
1720 LET D1=INT(F9/60)+1
1730 PRINT
1740 PRINT
1750 PRINT
1760 PRINT TAB(26); "FREQUENCY HISTOGRAM"
1770 PRINT
1780 PRINT "MIDPOINT";TAB(20);"FREQUENCIES"
1790 PRINT
1800 FOR I = 1 TO N1
1810 LET N2 = INT(F(I)/D1+.5)
1820 PRINT M(I);TAB(12);
```

(continued)

(program continued)

```
1830 FOR J = 1 TO N2
1840 PRINT "*";
1850 NEXT J
1860 PRINT
1870 NEXT I
1880 RETURN
1890 REM ........  END OF SUBROUTINE                    ...........
1900 REM
1910 REM ........    SUBROUTINE : SEQUENTIAL ............
1920 LET L = 1
1930 LET R = N
1940 LET S = 0
1950 IF R - L < 1 THEN 2250
1960 LET I = L
1970 LET J = R
1980 LET K = X(L)
1990 IF K>=X(J) THEN 2020
2000 LET J = J - 1
2010 GOTO 1990
2020 IF I<J THEN 2040
2030 GOTO 2140
2040 LET X(I) = X(J)
2050 LET X(J) = K
2060 LET I = I+1
2070 IF I<J THEN 2090
2080 GOTO 2140
2090 IF K>X(I) THEN 2060
2100 LET X(J) = X(I)
2110 LET X(I) = K
2120 GOTO 2000
2130 REM STACK THE DATA
2140 LET S = S + 1
2150 IF I - L < R - I THEN 2200
2160 LET L(S) = L
2170 LET R(S) = I - 1
2180 LET L = I+1
```

(continued)

(program continued)

```
2190 GOTO 1950
2200 LET L(S) = I + 1
2210 LET R(S) = R
2220 LET R = I - 1
2230 GOTO 1950
2240 REM FETCH FROM STACK
2250 IF S = 0 THEN 2300
2260 LET L = L(S)
2270 LET R = R(S)
2280 LET S = S - 1
2290 GOTO 1950
2300 RETURN
2310 REM ........  END OF SUBROUTINE              ...........
2320 REM
2330 REM ........    SUBROUTINE  MAXMIN ....................
2340 LET M1 = X(1)
2350 LET M2 = X(1)
2360 FOR J = 2 TO N
2370 IF M1 >X(J) THEN 2400
2380 IF M2 < X(J) THEN 2420
2390 GOTO 2440
2400 LET M1 = X(J)
2410 GOTO 2440
2420 LET M2=X(J)
2430 NEXT J
2440 RETURN
2450 REM ....      END OF SUBROUTINE       .............
2460 DATA 29
2480 DATA 69,10,30,88,20,80,75,60,70,40,45,55,88,99,100,71
2490 DATA 65,68,65,92,83,78,45,93,43,49,69,76
2500 END
```

The output of this program is as follows:

```
                 OUTPUT 8.3

DO YOU WANT INSTRUCTIONS ? YES
THIS PROGRAM CREATES A FREQUENCY TABLE AND GRAPHS
FREQUENCIES OF DATA. YOU CAN ENTER DATA EITHER
THROUGH AN INPUT STATEMENT OR TYPE IT IN THE DATA
STATEMENT STARTING AT LINE 2460.
INSTRUCTIONS : WHEN REQUESTED, ENTER THE FOLLOWING
               INFORMATION
  1)  THE NUMBER OF DATA POINTS (OBSERVATIONS)
  2)  ANSWER 'YES' IF THE DATA HAS ALREADY BEEN
      TYPED IN THE DATA STATEMENT. ANSWER 'NO', IF
      THE DATA IS NOT TYPED.
  3)  THE DATA MAY BE ENTERED IN ANY ORDER INTO
      THE DATA STATEMENT. HOWEVER, IF YOU WISH
      TO SEE THE DATA ORDERED IN DESCENDING ORDER,
      TYPE 'YES' WHEN PROMPTED.
IF THE NUMBER OF OBSERVATIONS (DATA POINTS) IS GREATER
THAN 500 OR THE NUMBER OF CLASS INTERVALS EXCEEDS 50,
THEN CHANGE THE DIM STATEMENT IN LINE 20 TO THE SIZE
YOU WANT.

HAVE YOU TYPED THE DATA IN THE DATA STATEMENT ? YES
DO YOU WANT TO SEE THE DATA IN SEQUENTIAL ORDER? YES
YOUR DATA IN SEQUENTIAL DATA IS

10  20  30  40  43  45  45  49  55  60  65  65  68  69  69  70  71
72  75  76  78  80  83  88  88  92  93  99  100

SMALLEST VALUE IS   10
LARGEST VALUE IS   100
RANGE IS  90

DO YOU WANT FREQUENCY TABLE ? YES
ENTER THE NUMBER OF FREQUENCY INTERVALS
IT COULD BE A NUMBER BETWEEN 5 AND 10? 5
```

OUTPUT 8.3 (continued)

FREQUENCY DISTRIBUTION

CLASS INTERVALS			MIDPOINTS	FREQ.	CUM. FREQ.	REL. FREQ.	REL. CUM FREQ.
10	-	28	19	2	2	0.069	0.069
28	-	46	37	5	7	0.172	0.241
46	-	64	55	3	10	0.103	0.345
64	-	82	73	12	22	0.414	0.759
82	-	100	91	7	29	0.241	1.000
TOTALS				29		1	

```
DO YOU WANT A FREQUENCY HISTOGRAM? YES

                    FREQUENCY HISTOGRAM

MIDPOINT                  FREQUENCIES

 19              **
 37              *****
 55              ***
 73              ************
 91              *******
```

CHAIN Statement

Another method of module programming is to create separate programs and then link them together by the **CHAIN** statement. The CHAIN statement instructs the BASIC program to stop executing the current program at the point of the CHAIN statement, to load another program named in the CHAIN statement and to start executing it. The general form of

the CHAIN statement is:

```
Ln  CHAIN   "program name", Ln1
```

This statement consists of the line number (Ln), the keyword CHAIN, and the name of the program within the quotes to be executed; and Ln1 is the line number in the chained program to be executed. If the Ln1 is not specified, the chained program is executed from the beginning. For example,

```
200 CHAIN "SORT"
```

If this statement is in a BASIC program, upon the execution of this statement, the computer will load the program named SORT and will execute that program from the beginning. However, if you intend to start the execution of the program at line 200, the statement would look as follows:

```
200   CHAIN "SORT", 200
```

In this statement, the computer loads the SORT program and starts the execution at line number 200. In order to call a program with the CHAIN, the program must exist somewhere on the disk.

During the chaining process, the original program is replaced by the new program. Therefore, all variables defined in the old program are erased. However, if you want to transfer the values of the defined variables to the new program, simply add the word ALL at the end of the CHAIN statement. For example,

```
200  CHAIN "SORT", ALL
```

or

```
200  CHAIN "SORT",200, ALL
```

Some of the BASIC versions do not use the ALL as a parameter in the CHAIN statement to transfer values to the new program. In that case, check the BASIC User Manual for an appropriate statement for transferring values. However, another most commonly used statement for this purpose is the COMMON statement. The general form of the common statement is:

```
Ln COMMON  var1, var2, ....
```

This statement consists of a line number (Ln), the keyword COMMON, and the list of variables which are defined in the program but to be transferred to the new program. For example,

```
 10  COMMON   A, B
 20  READ  A,B
        .
        .
        .
190  DATA 100,.10
200  CHAIN "SORT"
```

```
 10  REM PROGRAM SORT
        .
        .
        .
 90  PRINT A,B
100  END
```

In this example, the variables A and B are read in one program and transferred to another program.

The following programs are complete programs showing the use of the CHAIN statement.

PROGRAM 8.4

```
10 REM ........    PROGRAM : INSTR .........
20 REM
30 DIM X(500),F(50),M(50),B(50),C(50)
40 REM X() IS FOR READ DATA, F(I) IS FOR FREQUENCIES
50 REM C() IS FOR CUMULATIVE FREQUENCIES
60 REM M() IS MIDPOINTS OF CLASS INTERVALS
70 REM R() IS FOR CLASS LIMITS
80 LET B1$="PLEASE ANSWER YES OR NO"
90 PRINT "DO YOU WANT INSTRUCTIONS";
100 INPUT D$
110 IF D$="NO" THEN 160
120 IF D$="YES" THEN 150
130 PRINT B1$
140 GOTO 100
150 GOSUB 340
160 Z$="NO"
160 C1=0
170 PRINT
180 PRINT " HAVE YOU TYPED THE DATA IN THE DATA STATEMENT ";
190 INPUT A$
200 IF A$="YES" THEN 260
210 IF A$="NO" THEN PRINT DO YOU WANT TO INPUT THE DATA ";
220 INPUT Z$
230 IF Z$="NO" THEN GOSUB 340
240 PRINT
250 PRINT "ENTER THE NUMBER OF DATA POINTS ";
260 IF Z$="NO" THEN READ N   ELSE INPUT N
270 IF Z$="YES" THEN PRINT "ENTER ONE NUMBER AT A TIME"
280 FOR I=1 TO N
290 IF Z$="NO" THEN READ X(I)  ELSE  INPUT X(I)
```

(continued)

(program continued)

```
300 LET C1=C1+1
310 NEXT I
330 CHAIN "SEQ",ALL
340 PRINT "THIS PROGRAM CREATES A FREQUENCY TABLE AND GRAPHS"
350 PRINT "FREQUENCIES OF DATA. YOU CAN ENTER DATA EITHER "
360 PRINT "THROUGH AN INPUT STATEMENT OR TYPE IT IN THE DATA"
370 PRINT "STATEMENT STARTING AT LINE 2460."
380 PRINT "  "
390 PRINT "INSTRUCTIONS : WHEN REQUESTED, ENTER THE FOLLOWING"
400 PRINT "                 INFORMATION"
410 PRINT "  1)  THE NUMBER OF DATA POINTS (OBSERVATIONS)"
420 PRINT "  2)  ANSWER 'YES' IF THE DATA HAS ALREADY BEEN"
430 PRINT "      TYPED IN THE DATA STATEMENT. ANSWER 'NO', IF"
440 PRINT "      THE DATA IS NOT TYPED."
450 PRINT "  3)  THE DATA MAY BE ENTERED IN ANY ORDER INTO"
460 PRINT "      THE DATA STATEMENT. HOWEVER, IF YOU WISH"
470 PRINT "      TO SEE THE DATA ORDERED IN DESCENDING ORDER,"
480 PRINT "      TYPE 'YES' WHEN PROMPTED."
490 PRINT"IF THE NUMBER OF OBSERVATIONS(DATA)IS GREATER"
500 PRINT"THAN 500 OR THE NUMBER OF CLASS INTERVALS EXCEEDS50"
510 PRINT"THEN CHANGE  DIM STATEMENT IN LINE 20 TO THE SIZE"
520 PRINT"YOU WANT. THE PROGRAM REQUIRES DATA. PLEASE RUN THE"
530 PRINT "AGAIN ACCORDING TO THE INSTRUCTIONS."
540 IF Z$="NO" THEN END
550 DATA 29
560 DATA 69,10,30,88,20,80,75,60,70,40,45,55,88,99,100,71
570 DATA 65,68,65,92,83,78,45,93,43,49,69,76
580 END
```

PROGRAM 8.5

```
10 REM ........  PROGRAM : SMALL  ................
20 DIM X(500),F(50),M(50),B(50),C(50)
30 LET M1=X(1)
40 LET M2=X(N)
50 REM DETERMINE THE VALUE
60 IF X(1)>= THEN 100
70 IF ABS(10*X(1))<X(N) THEN 100
80 LET X9=X(1)
90 GOTO 120
100 LET X9=X(N)
110 IF C1<N THEN 130
120 LET L=L-3
130 LET N9=INT(72/L)
140 LET N8=INT(N/N9)+1
150 FOR I=1 TO N8
160 FOR J=1 TO N9
170 IF N9*(I-1)+J>N THEN 200
180 NEXT J
190 NEXT I
200 PRINT
210 PRINT "SMALLEST VALUE IS ";M1
220 PRINT "LARGEST VALUE IS  ";M2
230 PRINT "RANGE OF THE DATA IS ";M2-M1
240 PRINT
250 PRINT "DO YOU WANT FREQUENCY TABLE ";
260 INPUT Y$
270 IF Y$="YES" THEN CHAIN "FREQ"
280 END
```

PRROGRAM 8.6

```
 10 REM ........     PROGRAM : FREQ    ..........
 20 DIM X(500),F(50),M(50),B(50),C(50) 20 REM
 30 PRINT " ENTER THE NUMBER OF FREQUENCY INTERVALS."
 40 PRINT " IT COULD BE A NUMBER BETWEEN 5 AND 10"
 50 INPUT N1
 60 D = (M2-M1)/N1
 70 B(1) = M1
 80 REM TO SET UP LIMITS
 90 FOR I = 2 TO N1+1
100 LET B(I) = B(I-1)+D
110 NEXT I
120 REM TO DETERMINE CLASS FREQUENCIES
130 FOR I = 1 TO N
140 FOR J = 2 TO M1 + 1
150 IF X(I)>B(J) THEN 180
160 LET F(J-1)=F(J-1)+1
170 GOTO 190
180 NEXT J
190 NEXT I
200 REM  DETERMINES CLASS MIDPOINTS AND CUMULATIVE FREQUENCIES
210 FOR I = 1 TO N1
220 LET M(I) = B(I)+B(I+1))/2
230 LET C(I) = C(I-1) + F(I)
240 NEXT I
250 GOSUB 520
260 PRINT "DO YOU WANT A FREQUENCY HISTOGRAM";
270 INPUT D$
280 IF D$="NO" THEN 910
290 IF D$="YES" THEN GOSUB 310
300 GOTO 910
310 REM
320 REM ........     SUBROUTINE : FREQUENCY       ............
330 GOSUB 750
```

(continued)

(program continued)

```
340 LET D1=INT(F9/60)+1
350 PRINT
360 PRINT
370 PRINT
380 PRINT TAB(26); "FREQUENCY HISTOGRAM"
390 PRINT
400 PRINT "MIDPOINT";TAB(20);"FREQUENCIES"
410 PRINT
420 FOR I = 1 TO N1
430 LET N2 = INT(F(I)/D1+.5)
440 PRINT M(I);TAB(12);
450 FOR J = 1 TO N2
460 PRINT "*";
470 NEXT J
480 PRINT
490 NEXT I
500 RETURN
510 REM ........  END OF SUBROUTINE                   ..............
520 REM ........   SUBROUTINE : PRINTING       ...............
530 PRINT
540 PRINT
550 REM FRENQEUNCY TABLE
560 PRINT TAB(26);"FREQUENCY DISTRIBUTION"
570 PRINT
580 PRINT TAB(43):"CUM";TAB(52);"REL";TAB(64);"REL CUM"
590 PRINT TAB(5);"CLASS LIMITS";
600 PRINT TAB(22);"MIDPOINTS";TAB(34);"FREQ";TAB(43);
610 PRINT "FREQ";TAB(52); "FREQ";TAB(64); "FREQ";
620 PRINT "-----------------------------------------------------"
630 PRINT "---------------------"
640 FOR I = 1 TO N1
650 PRINT B(I);TAB(10);" -  ";B(I+1);
660 PRINT TAB(24);M(I);TAB(34);F(I);TAB(43);C(I);
670 PRINT TAB(50);
680 PRINT USING"##.##       ###.###";F(I)/N,C(I)/N
690 NEXT I
```

(continued)

(program continued)

```
700 PRINT "--------------------------------------------------"
710 PRINT "--------------------"
720 PRINT "TOTALS";TAB(34);C(N1);TAB(50);C(N1)/N
730 RETURN
740 REM ........  END OF SUBROUTINE                 .............
750 REM ........    PROGRAM  MAXMIN ...................
760 LET M1 = X(1)
770 LET M2 = X(1)
780 FOR J = 2 TO N
790 IF M1 >X(J) THEN 820
800 IF M2 < X(J) THEN 840
810 GOTO 850
820 LET M1 = X(J)
830 GOTO 850
840 LET M2=X(J)
850 NEXT J
860 RETURN
870 REM ....       END OF SUBROUTINE                 ............
880 END
```

PROGRAM 8.7

```
 10 REM ........     PROGRAM : SEQ ............
 30 DIM X(500),F(50),M(50),B(50),C(50)
 30 LET L = 1
 40 LET R = N
 50 LET S = 0
 60 IF R - L < 1 THEN 350
 70 LET I = L
```

(continued)

(program continued)

```
 80 LET J = R
 90 LET K = X(L)
100 IF K>=X(J) THEN 120
110 LET J = J - 1
120 GOTO 90
130 IF I<J THEN 140
140 GOTO 240
150 LET X(I) = X(J)
160 LET X(J) = K
170 LET I = I+1
180 IF I<J THEN 190
190 GOTO 240
200 IF K>X(I) THEN 140
210 LET X(J) = X(I)
220 LET X(I) = K
230 GOTO 100
240 REM STACK THE DATA
250 LET S = S + 1
260 IF I - L < R - I THEN 300
270 LET L(S) = L
280 LET R(S) = I - 1
290 LET L = I+1
300 GOTO 50
310 LET L(S) = I + 1
320 LET R(S) = R
330 LET R = I - 1
340 GOTO 50
350 REM FETCH FROM STACK
360 IF S = 0 THEN 400
370 LET L = L(S)
380 LET R = R(S)
390 LET S = S - 1
400 GOTO 50
410 PRINT "DO YOU WANT TO SEE THE DATA IN ASCENDING ORDER";
420 INPUT B$
```

(continued)

(program continued)

```
430 IF B$="NO" THEN CHAIN "SMALL"
440 IF B$="YES" THEN 460
450 PRINT B1$
460 GOTO 410
470 PRINT
480 PRINT "YOUR DATA IN SEQUENTIAL ORDERED IS "
490 PRINT
500 REM  **  TO PRINT ORDERED DATA
510 FOR I = 1 TO N
520 PRINT X(I);
530 NEXT I
540 CHAIN "SMALL"
550 END
```

These programs have to be entered and saved as separate files (programs). We need to load and run the main program only. The main program then calls the other programs when needed. The output of these program will be same as shown in Output 8.3.

QUESTIONS

1. Define a function. What are the differences between a built-in function and a user-defined function?

2. What are mathematical built-in functions used for?

3. Show the results of the following mathematical functions:

```
SQR (100)
FIX ( 45.79)
INT ( 6.99)
SGN (-9098)
```

4. Write a program that generates 100 3-digit random integers (fixed-decimal or whole) numbers.

5. What does the statement RANDOMIZE do?

6. Show the results of the following string functions:

```
LEFT$  ("PESHAWAR", 6)
RIGHT$ ("PESHAWAR", 6)
MID$   ("PUNJAB", 3, 3)
LTRM$  ("    SIND   ")
RTRM$  ("    SIND   ")
LPAD$  ("SIND", 10)
RPAD$  ("BALUCHISTAN", 15)
RPAD$  ("NAME", 3)
LEN    ("BASIC")
POS    ("BALUCHISTAN", "T", 5)
UPRC$  ("pakistan")
LWRC$  ("PAKISTAN")
```

7. Correct errors, if any, in the following user-defined functions.

a)
```
DEF FNP (P1) = P + P *.03
 .
READ X
B    =   FNP (X)
```

b)
```
DEF FND (P1) = P1 + P1 * .03

READ  X
B     =  FND (X)
```

c)
```
DEF FND

READ X
B    = FND (X)
```

8. What is a subroutine? What are the differences between a subroutine and a user-defined function?

9. What is a main program? What is the purpose of a main program?

10. Write a program which reads names and gross pay for each employee, and prints the names, gross pay, retirement and income tax deductions, and net pay. However the program should use a subroutine to calculate the tax amount using the following table.

INCOME PER MONTH	TAX RATE
Rs. 0 - 1000	0 %
1001 - 2000	2
2001 - 3000	5
3001 - 4000	10
4001 - 5000	15
5001 - 8000	25
8001 and over	35

Add another subroutine to calculate the retirement deductions using the following table.

INCOME PER MONTH	RETIREMENT RATE
Rs. 0 - 1000	0 %
1001 - 2000	1
2001 - 3000	2
3001 - 4000	3
4001 - 5000	4
5001 - 8000	5
8001 and over	6

However, once a person has had deducted a total of Rs. 5,000 per year for the retirement fund, there should be no more deduction for that year.

11. Write subroutines, one for each, for calculating a mean, a median, a standard deviation, and a frequency table of grades of any number of students, but not to exceed 200.

12. What is the chain statement used for?

13. What are the differences between a subprogram and a subroutine?

14. Write subprograms using Question 10.

15. Write subprograms using Question 11.

16. Write a subprogram that sorts the data of any program.

17. Correct errors, if any, in the following program segments.

a)
```
10 A1 = 90
20 END
30 GOSUB 400
40 IF A1> 90 THEN PRINT "A"
.
.
.
60 RETURN
90 END
```

b)
```
10 A1 = 90
20 GOSUB 40
30 END
40 IF A1> 90 THEN PRINT "A"
.
.
.
60 RETURN
90 END
```

c)
```
10 A1 = 90
20 GOSUB 40
40 IF A1> 90 THEN PRINT "A"
.
.
.
60 RETURN
90 END
```

18. Write a program that reads salespersons' names, amount sold, and hours worked during a week. Write a subroutine that calculates the commission based on the following table.

HOURS WORKED PER WEEK	AMOUNT SOLD PER WEEK	COMMISSION RATE AND BASE PAY	PLUS OVERTIME RATE ON THE BASE PAY
0 - 20	Rs. 1000 - 2000	10 % plus Rs. 500	
	2001 - 3000	15 plus Rs. 500	
	3001 - 5000	20 plus Rs. 500	
	5000 - 8000	25 plus Rs. 500	
	8000 and over	30 plus Rs. 500	
21 - 40	Rs. 1000 - 2000	5 % plus Rs. 1000	
	2001 - 3000	8 plus Rs. 1000	
	3001 - 5000	11 plus Rs. 1000	
	5000 - 8000	14 plus Rs. 1000	
	8000 and over	16 plus Rs. 1000	
41 - 50	Rs. 1000 - 2000	5 % plus Rs. 1000	
	2001 - 3000	8 plus Rs. 1000	
	3001 - 5000	11 plus Rs. 1000	
	5000 - 8000	14 plus Rs. 1000	
	8000 and over	16 plus Rs. 1000	
51 and over	Rs. 1000 - 2000	5 % plus Rs. 1000	5 %
	2001 - 3000	8 plus Rs. 1000	6
	3001 - 5000	11 plus Rs. 1000	7
	5000 - 8000	14 plus Rs. 1000	8
	8000 and over	16 plus Rs. 1000	9

Write another subroutine that prints all the information that is read (inputted) and the calculated commission amount, the base pay, and the total pay.

19. Write subprograms using Question 18.

20. What is the Common statement, and when will you use it?

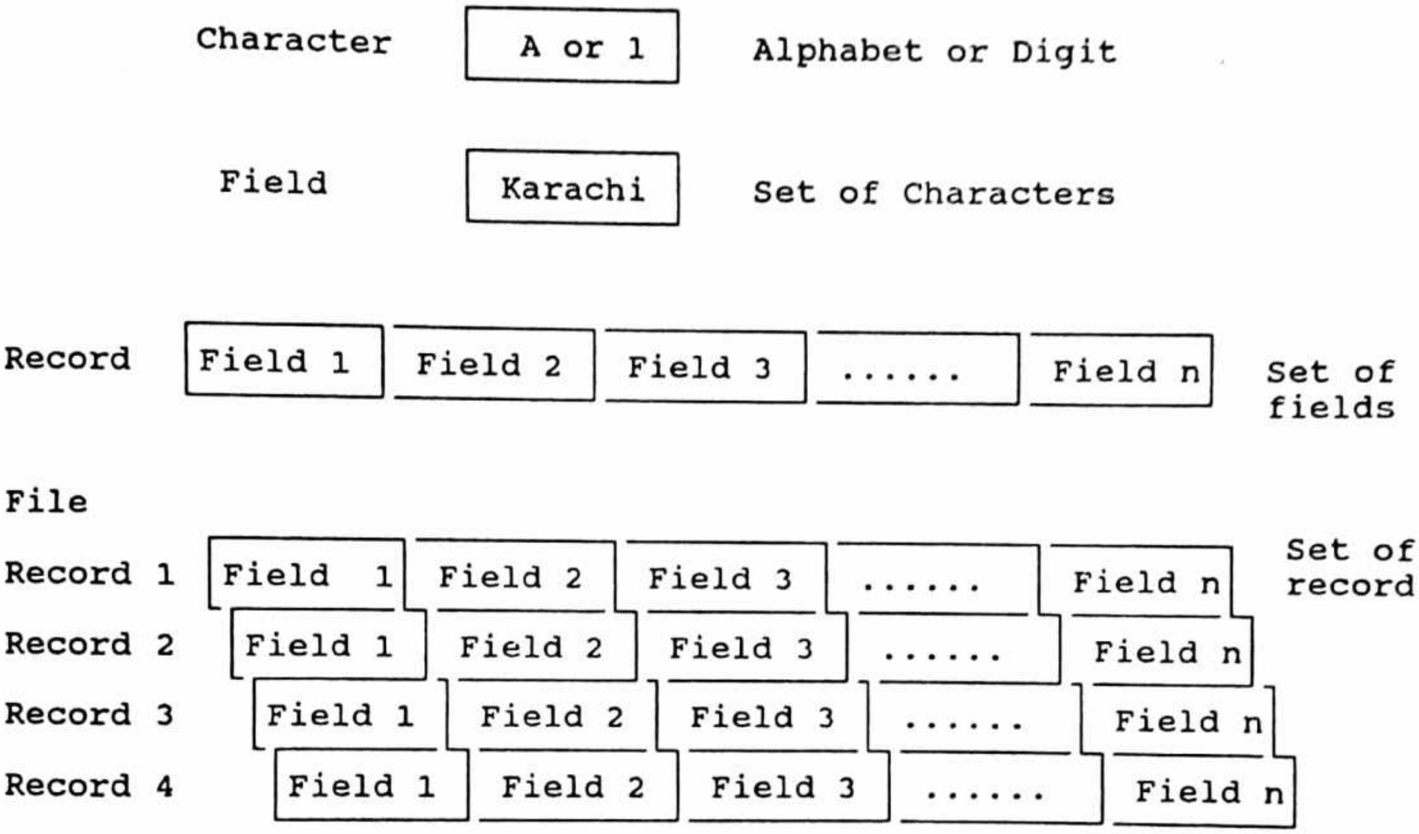
CHAPTER 9

FILE MANIPULATION

INTRODUCTION - FILE

So far we read and processed data as a part of a program. But after the program was run, the same data or the processed data was not available for other programs to process. That is, every time a program was run, the data was read from the DATA statement or inputted every time. In business or non-business situations, it is not feasible to enter a large quantity of data for each program or to be made a part of a program to be read every time the program is executed. Further, the same data may be needed by other programs. Therefore, the same data must be entered or saved as a part of each program. This is either economical nor convenient. Therefore, large quantities of data needed by programs should be stored separately on a secondary storage device (such as a disk or diskette or tape). It does not matter whether the stored data is entered initially or generated by a program. It is economical and convenient to store data on a storage device. Data stored on a storage device is stored under a file name. A file is a collection of related records. A record consists of related fields. A field is a set of characters (alphabets and/or digits). The following figure shows the relationship between characters and field, fields and record, and records and file.

FIGURE 9.1



For example, the following records make up a file. The file contains information about students. That is, each record contains information about a student. There are fifteen records in this file. Each record consists of six fields. The field names are : ID, FIRST NAME, AGE, MARITAL STATUS, MAJOR, and CREDIT HOURS. Each field consists of characters or digits. For example, field ID consists of four digits, and field FIRST NAME consists of alphabets. Since the file contains records of the students, it will be named as STUDENT file.

TABLE 9.1

File Name : STUDENT

FIELD NAMES

RECORD	ID	FIRST NAME	AGE	MARITAL STATUS	MAJOR	CREDIT HOURS
1	7736	Bobbi	21	U	MIS	15
2	9753	Carol	20	M	MGT	18
3	4283	Connie	22	M	MIS	12
4	6699	Dianne	21	U	ACC	15
5	3750	Doug	22	M	MIS	12
6	4321	Jane	22	M	MGT	15
7	0123	Jeff	22	M	MIS	12
8	1111	John	22	M	MIS	12
9	1006	Julie	21	U	MGT	12
10	1122	Kim	22	M	MIS	12
11	2048	Kris	22	U	MIS	18
12	9110	Patrick	22	U	ACC	12
13	9462	Ricit	22	U	MIS	15
14	4461	Rod	29	U	ACC	06
15	5612	Susan	21	U	MIS	16

Marital Status codes: U = Unmarried M = Married
Major COdes : MIS = Management Information System
MGT = Management, ACC = Accounting

Usually, each record in a file has one field that uniquely identifies that record. This field is referred to as a **primary key field.** The purpose of a primary key field is to retreive a specified record in the file. For example, the ID numbers in the ID field are different in each record. Therefore, the ID field will be used as a primary key field in this file.

Data stored in a file is written and retrieved one record at a time. Records in a file can be stored sequentially or non-sequentially. However, a sequential file requires sorting records before storing them. That is,

records in a sequential file are stored one after another in the sequential order of the key. Therefore, to access a record, the computer has to read each record one after the other until it finds the record you want. The advantage of a sequential file is that all records can be listed in a sequence of the primary key.

In a consequential file, records are stored in any order. However, to access a record requires the same procedure as in the sequential file. That is, each record has to be read and compared with the key of the record you want. Another form of nonsequential record is a random file. In a random file, records are identified by numbers(address). One of the methods for creating a random file is to number the records as they are entered. The first record entered is number 1, the second record is number 2, and so on. However, some BASICs cannot create and access a random file. To create a random access requires a different method.

There are three operations which may be performed on a data file:

1. Creating a file (storing a file on a disk)
2. Updating a file (adding/deleting/changing records in an existing file)
3. Accessing a file (locating records in a file)

FILE STORAGE : Sequential and Nonsequential File

Creating a file requires these steps:

1. Opening a file requires informing the computer about the name of the file. The computer needs this name to create a space on a disk for storing the data. There is a special form to the statement for opening a file.

2. Writing a record into the file requires that we input value for each field in a record, and then tell the computer to write it in the file. There is a special form to the statement for writing a record.

3. Closing a file also requires a special statement. When writing records into a file is completed, you must close the file.

Reading a file requires these steps:

1. Opening a file requires informing the computer about the name of the file. The computer needs this name to locate the storage space on the disk where the file was originally stored. There is a special form to the statement that opens a file.

2. Reading a record requires that we read all fields in a record (the whole record), and then process each field. There is a special form to the statement for reading a record.

3. Closing a file requires a special statement. When you have finished reading records from a file, you must close the file.

We will be following all these steps in this chapter.

OPEN Statement

Since BASIC is meant to function in an interactive environment, files will be stored on a disk or a diskette. Files can be created through a program or created by inputting data directly from the keyboard. Regardless of the manner of data input, the computer has to be informed that a file is being created. The general form of the statement is:

```
Ln  OPEN  "filename" FOR INPUT or OUTPUT AS FILE #channel number
```

This statement consists of the line number (Ln), the Keyword OPEN followed by the file name within quotes. The filename is a string of up to 8 characters. The file name is followed by the keywords FOR and INPUT or OUTPUT. The INPUT is used when you are reading a file, and the OUTPUT is used when you are creating (writing) a file. The INPUT or OUTPUT is followed by the keywords AS FILE. The word FILE is followed by the number sign (#) and a number between one and ten which is then used later in the program to refer to the file. The number once assigned to a file in a program would be assigned to that file until the file is closed or cleared from the primary memory. For example, to create a file, the OPEN statement could be:

```
10  OPEN   "student" FOR OUTPUT AS FILE #1
```

In this statement, line number 10 opens the file named Student for output and assigns the channel number (reference number) 1. We can only write data into this file since it is opened for output. However, a file should be opened with INPUT when you want to read a file.

There are many versions of BASIC and each version has its own statement for files. Refer to Table 9.2 for an appropriate OPEN statement for your system. Also, check your system's BASIC manual for an appropriate file statement. For example, the IBM personal computer (PC) BASIC uses the following statement for opening a file.

```
Ln OPEN "filename" FOR INPUT or OUTPUT AS channel number
```

In this statement, the statement number is followed by the keyword OPEN and the file name within quotes. The file name is followed by the keywords FOR and either INPUT or OUTPUT. The INPUT or OUTPUT word is followed the keywords AS FILE and the channel number without a number (#) sign.

For example, to open the Student file for output, the statement is:

```
10 OPEN  "student" FOR OUTPUT AS FILE 1
```

In this statement, the file Student is opened for output (writing).

Some other versions of the OPEN statement are listed in Table 9.2.

TABLE 9.2

OTHER VERSIONS OF OPEN STATEMENT

SYSTEM	STATEMENT
CDC BASIC	Ln OPEN FILE #number ="filename"
BASIC PLUS	Ln OPEN "filename" FOR INPUT or OUTPUT AS FILE #number
ANSI BASIC	Ln OPEN #number:NAME "filename", ACCESS INPUT or OUTPUT, ORGANIZATION SEQUENTIAL
Microsoft BASIC	Ln OPEN "I" or "O", #number, "filename"
Digital BASIC	Ln OPEN "filename" FOR INPUT or OUTPUT AS FILE #number
VS BASIC	Ln OPEN "filename" IN or OUT or ALL (When a file is to be used for both input and output, use ALL)

NOTE: When a word is capitalized, you must use that word exactly. In cases where the word "or" is indicated between words, you should choose one or the other keyword but not both.

CLOSE Statement:

All files opened in a program must be closed before the termination of the program. The general form of a statement for a closing a file is:

```
Ln  CLOSE  #channel number
```

This statement consists of the line number (Ln), the keyword CLOSE followed by the channel number. All files can be closed within one CLOSE statement by listing all the channel numbers associated with the opened files, or each file could have a separate close statement. For example,

```
10   CLOSE  #1
```

If you have opened more than one file, the following statement could be used to close all files.

```
100   CLOSE  #1, #2, #3
```

or

```
100   CLOSE #1
110   CLOSE #2
120   CLOSE #3
```

or

```
100   CLOSE
```

This last statement without a channel number will close all opened files.

However, for IBM's VS BASIC, the CLOSE statement is:

```
100 CLOSE "filename"      (IBM's VS BASIC)
```

In this statement, the name of file to be closed must be identified, and not the channel number.

CREATING A FILE : PRINT Statement

Once a file is opened for output, you may want to write data into it. The following is the general form of PRINT (Write) statement which will write data into an output file.

```
Ln  PRINT #channel number, variables list
```

or

```
Ln  WRITE #channel number, variables list
```

In this statement, the line number (Ln) is followed by the keyword PRINT (or WRITE for some systems) and a channel number and a comma. The comma is followed the names of the variables to be written into the file. For example, to write a record of the file in Table 9.1, the statement is:

```
10 PRINT #1,I,N$,A,S$,M$,C
```

or

```
10 WRITE #1,I,N$,A,S$,M$,C
```

This statement writes six variables, I (ID), N$ (NAME), A (AGE), S$ (MARITAL STATUS), M$ (MAJOR), C (CREDIT HOURS) for each record in the file assigned to channel number 1.

As mentioned previously, some BASICs might use a different form for writing data into a file. Refer to Table 9.3 for an appropriate PRINT (WRITE) statement for your system. Also, check your BASIC manual for an appropriate form of the statement. For example, the general form of PRINT statement for IBM PC is:

```
Ln  PRINT #1,var1;",";var2;",";var3
```

In this statement, each variable name except the last one is followed by a comma enclosed within quotes. In this statement, the comma is a delimiter. A delimiter indicates to the computer where the value of a variable ends. In addition, the variable names are separated by semicolon (;) rather than a comma. For example, to write a record of the file in Table 9.1, the statement is:

```
10 PRINT #1,I;",";N$;",";A;",";S$;",";M$;",";C
```

In this example, variable I is followed by a comma within quotes and so are the other variables except the last variable C.

Some other versions of the PRINT statement are listed in Table 9.3.

TABLE 9.3

OTHER VERSIONS OF THE **PRINT** STATEMENT

SYSTEM	STATEMENT
CDC BASIC	Ln PRINT #number, variables list
BASIC PLUS	Ln PRINT #number, variables list
ANSI BASIC	Ln PRINT #number: variables list
Microsoft BASIC	Ln PRINT #number, variables list
Digital BASIC	Ln PRINT #number, variables list
VS BASIC	Ln PUT "filename", variables list

NOTE: When a word is capitalized, you must use that word exactly. The variables list means the names of the fields to be written into the file.

CREATING A NON-SEQUENTIAL FILE

To show how to create a nonsequential file for the above student records in Table 9.1, let us write a program.

PROGRAM 9.1

```
 10  REM TO CREATE STUDENT FILE
 20  OPEN  "STUDENT"  FOR INPUT AS FILE #1
 30  FOR K = 1  TO  100
 40  PRINT "TYPE ZEROS FOR ALL VARIABLES TO STOP THE PROGRAM"
 50  PRINT "TYPE SIX VALUES, ONE FOR EACH VARIABLE: ID, "
 60  PRINT " NAME , AGE, MARITAL STATUS, MAJOR, CREDIT HOURS"
 70  INPUT I,N$,A,S$,M$,C
 80  IF I=0 THEN  110
 90  PRINT #1,I,N$,A,S$,M$,C
100  NEXT K
110  CLOSE #1
120  PRINT " YOUR FILE IS NOW CREATED"
130  END
```

In this program, statement 20 opens the file named STUDENT and assigns it the number 1. The loop starts in line 30. In line 70, the data is to be entered from the keyboard. Line 80 checks to see if zeros are entered. If there are zeros, the computer closes the file. As long as zeros are not entered, the computer asks for up to 100 records and writes the records into the file.

When this program is executed, the screen would look as follows.

OUTPUT 9.1

```
TYPE ZEROS FOR ALL SIX  VARIABLES TO STOP THE PROGRAM
TYPE SIX VALUES, ONE FOR EACH VARIABLE: ID,
 NAME , AGE, MARITAL STATUS, MAJOR, CREDIT HOURS
? 7736,Bobbi,21,U,MIS,15
TYPE ZEROS FOR ALL SIX  VARIABLES TO STOP THE PROGRAM
TYPE SIX VALUES, ONE FOR EACH VARIABLE: ID,
 NAME , AGE, MARITAL STATUS, MAJOR, CREDIT HOURS
? 9753,Carol,20,M,MGT,18
TYPE ZEROS FOR ALL SIX  VARIABLES TO STOP THE PROGRAM
TYPE SIX VALUES, ONE FOR EACH VARIABLE: ID,
  NAME , AGE, MARITAL STATUS, MAJOR, CREDIT HOURS
? 4283,Connie,22,M,MIS,12
TYPE ZEROS FOR ALL SIX  VARIABLES TO STOP THE PROGRAM
TYPE SIX VALUES, ONE FOR EACH VARIABLE: ID,
  NAME , AGE, MARITAL STATUS, MAJOR, CREDIT HOURS
? 6699,Dianne,21,U,ACC,15
TYPE ZEROS FOR ALL SIX  VARIABLES TO STOP THE PROGRAM
TYPE SIX VALUES, ONE FOR EACH VARIABLE: ID,
 NAME , AGE, MARITAL STATUS, MAJOR, CREDIT HOURS
? 3750,Doug,22,M,MIS,12
TYPE ZEROS FOR ALL SIX  VARIABLES TO STOP THE PROGRAM
TYPE SIX VALUES, ONE FOR EACH VARIABLE: ID,
  NAME , AGE, MARITAL STATUS, MAJOR, CREDIT HOURS
? 4321,Jane,22,M,MGT,15
TYPE ZEROS FOR ALL SIX  VARIABLES TO STOP THE PROGRAM
TYPE SIX VALUES, ONE FOR EACH VARIABLE: ID,
  NAME , AGE, MARITAL STATUS, MAJOR, CREDIT HOURS
? 0123,Jeff,22,M,MIS,12
TYPE ZEROS FOR ALL SIX  VARIABLES TO STOP THE PROGRAM
TYPE SIX VALUES, ONE FOR EACH VARIABLE: ID,
  NAME , AGE, MARITAL STATUS, MAJOR, CREDIT HOURS
? 1111,John,22,M,MIS,12
```

(continued)

OUTPUT 9.1 (continued)

```
TYPE ZEROS FOR ALL SIX  VARIABLES TO STOP THE PROGRAM
TYPE SIX VALUES, ONE FOR EACH VARIABLE: ID,
  NAME , AGE, MARITAL STATUS, MAJOR, CREDIT HOURS
? 1006,Julie,21,U,MGT,12
TYPE ZEROS FOR ALL SIX  VARIABLES TO STOP THE PROGRAM
TYPE SIX VALUES, ONE FOR EACH VARIABLE: ID,
  NAME , AGE, MARITAL STATUS, MAJOR, CREDIT HOURS
? 1122,Kim,22,M,MIS,12
TYPE ZEROS FOR ALL SIX  VARIABLES TO STOP THE PROGRAM
TYPE SIX VALUES, ONE FOR EACH VARIABLE: ID,
  NAME , AGE, MARITAL STATUS, MAJOR, CREDIT HOURS
? 0,0,0,0,0,0

YOUR FILE IS NOW CREATED
```

READING A FILE : INPUT Statement

Once a file is created, it can be read later in the same program or by other programs. However, all files to be read or to be written must be opened first. Once opened for input, to read a file, the general statement for reading a record in a file is:

1) Ln INPUT #channel number, variables list

or

2) Ln GET "filename", variables list (IBM's VS BASIC)

In statement 1, the line number (Ln) is followed by a keyword INPUT and a channel number. The channel number is followed by a comma and a list of variables (fields) in the record. For example, to read a record in the above file:

```
30 INPUT #1, I,N$,A,S$,M$,C
```

This statement reads the six fields (I,N$,A,S$,M$,C) (whole record) in the file assigned to channel number 1.

Let us create a program that reads and prints all the records in the file we created.

PROGRAM 9.2

```
10   REM TO READ THE STUDENT FILE
20   OPEN  "student" FOR INPUT AS  #1
30   FOR I = 1  TO  100
40   INPUT #1, I,N$,A,S$,M$,C
50   PRINT I,N$,A,S$,M$,C
60   NEXT I
70   CLOSE #1
80   END
```

Notice that line 20 in the above program opens the file created in Program 9.1 for input. Line 40 reads the six fields (whole record) in each record, and line 50 prints them. However, once there are no more records in the file, the computer will print an error message indicating the End of File (EOF) and will stop executing the program. This is an abnormal termination of the file. For example, the output of the program is as follows:

OUTPUT 9.2

```
7736,Bobbi,21,U,MIS,15
9753,Carol,20,M,MGT,18
4283,Connie,22,M,MIS,12
6699,Dianne,21,U,ACC,15
3750,Doug,22,M,MIS,12
4321,Jane,22,M,MGT,15
0123,Jeff,22,M,MIS,12
1111,John,22,M,MIS,12
1006,Julie,21,U,MGT,12
1122,Kim,22,M,MIS,12
INPUT PAST END IN LINE 40
```

In this output, the last line is an error message indicating that there is no data available to be read. In order to avoid the abnormal termination, it is better to check for the end of the file. The format of the statement is:

```
Ln  IF END #channel number THEN  statement number
```

In this statement, the line number is followed by the keywords IF and END and a channel number. The channel number is followed by a statement number where the program will be transferred when the end of the file is reached. For example,

```
40 IF END #1  THEN 90
```

In this statement, the keywords IF and END are followed by the channel number 1 and the channel number is followed by the keyword THEN and the statement number 90. That is, when the file runs out of data, the program will transfer to statement 90.

As indicated previously, not all BASICs follow the same format. Check your BASIC manual for an appropriate statement. For example, the IBM PC uses the following format:

```
Ln IF EOF(1) THEN  statement number
```

In this statement, the keyword IF is followed by the function EOF(1). The EOF function checks the file and when the file reaches the end, the computer returns the value of 1 which makes the condition true and the IF-THEN condition is satisfied which causes the program to transfer to the specified statement number.

Let us rewrite the above program to prevent an abnormal termination of the program.

PROGRAM 9.3

```
10   REM TO READ AND CHECK THE END OF FILE
20   OPEN  "student" FOR INPUT AS FILE #1
30   FOR I = 1  TO  100
40   IF END #1 THEN  80
50   INPUT #1, I,N$,A,S$,M$,C
60   PRINT 1,I,N$,A,S$,M$,C
70   NEXT I
80   CLOSE #1
90   END
```

The output of the above program is as follows:

OUTPUT 9.3

```
7736,Bobbi,21,U,MIS,15
9753,Carol,20,M,MGT,18
4283,Connie,22,M,MIS,12
6699,Dianne,21,U,ACC,15
3750,Doug,22,M,MIS,12
4321,Jane,22,M,MGT,15
0123,Jeff,22,M,MIS,12
1111,John,22,M,MIS,12
1006,Julie,21,U,MGT,12
1122,Kim,22,M,MIS,12
```

CREATING A SEQUENTIAL FILE

For creating a sequential file, records are entered as usual. But before they are stored in a file, they must be sorted by a primary key. In order to create a sequential file, using the same data as entered in OUTPUT 9.1, let us write a program which reads and then sorts the data before storing it in the file. For example,

PROGRAM 9.4

```
 10 REM TO CREATE STUDENT SEQUENTIAL  FILE
 20 REM TO CREATE AN ARRAY FOR SORTING DATA
 30 DIM Z(20),W$(20),B(20),H$(20), D$(20),G(20)
 40 FOR K = 1  TO  100
 50 PRINT "TYPE ZEROS FOR ALL VARIABLES TO STOP THE PROGRAM"
 60 PRINT "TYPE SIX  VALUE, ONE FOR EACH VARIABLE: ID,"
 70 PRINT "NAME, AGE, MARITAL STATUS, MAJOR, CREDIT HOURS"
 80 INPUT I(K),N$(K),A(K),S$(K),M$(K),C(K)
 90 IF I(K)=0 THEN  100
100 NEXT K
110 OPEN "STUDENT" FOR OUTPUT AS FILE #1
120 REM TO SORT RECORDS BEFORE WRITING THEM
130 REM TO FIND RECORD WITH THE SMALLEST ID NUMBER
140 FOR J= 1 TO K-1
150 K1=I(J)
160 L = J
170 FOR R = J TO K-1
180 IF K1>I(R) THEN L=R
190 IF K1>I(R) THEN K1=I(R)
200 NEXT R
210 PRINT #1,I(L);N$(L);A(L);S$(L);M$(L);C(L)
220 REM TO REARRANGE ARRAY BY MOVING SMALLEST ID TO THE TOP
230 A1=I(J)
240 Q$=N$(J)
250 V =A(J)
```

(continued)

PROGRAM 9.4 (continued)

```
260 T$=S$(J)
270 U$=M$(J)
290 X = C(J)
300 I(J)=I(L)
310 N$(J)=N$(L)
320 A(J)=A(L)
330 S$(J)=S$(L)
340 M$(J)=M$(L)
350 C(J)=C(L)
360 I(L)=A1
370 N$(L)=Q$
380 A(L)=V
390 S$(L)=T$
400 M$(L)=U$
410 C(L)=X
420 NEXT J
430 CLOSE #1
430 END
```

In this program, the data is entered and stored in arrays and then sorted before being written into the file. Now if the file is read by the program, the output would look as follows:

OUTPUT 9.4

```
0123,Jeff,22,M,MIS,12
1006,Julie,21,U,MGT,12
1111,John,22,M,MIS,12
1122,Kim,22,M,MIS,12
2048,Kris,22,U,MIS,18
3750,Doug,22,M,MIS,12
4283,Connie,22,M,MIS,12
4321,Jane,22,M,MGT,15
6699,Dianne,21,U,ACC,15
7736,Bobbi,21,U,MIS,15
9753,Carol,20,M,MGT,18
```

Notice that the records are listed in the order of the ID numbers.

UPDATING A FILE : APPEND Statement

After you create a file, it may be necessary to add records to the same file. However, once you close the file and open it again for output, new records are written at the beginning of the file, deleting the old records. That is, it creates a new file. Therefore, to add additional records to the end of the file, you must inform the computer to open the file for output and move to the end of the file. The general form of the statement is:

```
1) Ln  OPEN  "filename"  FOR APPEND AS FILE #channel number
```

or

```
2) Ln  OPEN  "filename" OUTPUT REUSE          (IBM's VS BASIC)
```

In statement 1, the keyword APPEND is used. The file is opened for output, and the file name is followed by the keyword APPEND indicating that you want to add records to the end of the file. For example,

```
10  OPEN "student", FOR APPEND AS FILE #1
```

Let us write a program to add additional records to the STUDENT file.

PROGRAM 9.5

```
 10  REM TO UPDATE THE STUDENT FILE
 20  OPEN  "STUDENT" FOR APPEND AS FILE #1
 30  FOR I = 1  TO  100
 40  PRINT "TYPE ZEROS FOR ALL VARIABLES TO STOP THE PROGRAM"
 50  PRINT "TYPE SIX VALUES, ONE FOR EACH VARIABLE: ID, "
 60  PRINT " NAME , AGE, MARITAL STATUS, MAJOR, CREDIT HOURS"
 70  INPUT I,N$,A,S$,M$,C
 80  IF I=0 THEN  110
 90  PRINT #1,I,N$,A,S$,M$,C
100  NEXT I
110  CLOSE #1
120  END
```

In this program, statement 20 opens the file named STUDENT for output and assigns it the number 1. Line 80 checks to see if zeros are entered. If there are zeros, the computer closes the file and stops the program. As long as zeros are not entered, the computer asks for more records and writes the records at the end of the file.

When this program is executed, the screen would look as follows.

(next page)

OUTPUT 9.5

```
TYPE ZEROS FOR ALL FIVE VARIABLES TO STOP THE PROGRAM
TYPE SIX VALUES, ONE FOR EACH VARIABLE: ID,
  NAME , AGE, MARITAL STATUS, MAJOR, CREDIT HOURS
? 2048,Kris,22,U,MIS,18
TYPE ZEROS FOR ALL FIVE VARIABLES TO STOP THE PROGRAM
TYPE SIX VALUES, ONE FOR EACH VARIABLE: ID,
  NAME , AGE, MARITAL STATUS, MAJOR, CREDIT HOURS
? 9110,Patrick,22,U,ACC,12
TYPE ZEROS FOR ALL FIVE VARIABLES TO STOP THE PROGRAM
TYPE SIX VALUES, ONE FOR EACH VARIABLE: ID,
  NAME , AGE, MARITAL STATUS, MAJOR, CREDIT HOURS
? 9462,Ricit,22,U,MIS,15
TYPE ZEROS FOR ALL FIVE VARIABLES TO STOP THE PROGRAM
TYPE SIX VALUES, ONE FOR EACH VARIABLE: ID,
  NAME , AGE, MARITAL STATUS, MAJOR, CREDIT HOURS
? 4461,Rod,29,U,ACC,06
TYPE ZEROS FOR ALL FIVE VARIABLES TO STOP THE PROGRAM
TYPE SIX VALUES, ONE FOR EACH VARIABLE: ID,
  NAME , AGE, MARITAL STATUS, MAJOR, CREDIT HOURS
? 5612,Susan,21,U,MIS,16
TYPE ZEROS FOR ALL FIVE VARIABLES TO STOP THE PROGRAM
TYPE SIX VALUES, ONE FOR EACH VARIABLE: ID,
  NAME , AGE, MARITAL STATUS, MAJOR, CREDIT HOURS
? 0,0,0,0,0,0
```

The BASIC statement used by IBM PC for adding data into existing file is as follows:

```
Ln OPEN "filename" FOR  APPEND AS channel number
```

In this statement, the keyword APPEND is also used instead of INPUT or OUTPUT.

SEARCHING A FILE : INPUT and IF-THEN Statement

To access a certain record in the file, we need a program which will ask for the value of a field in the record and then search each record for that value until the record is found. For example, the following program will allow you to access any specified stored record in the file.

PROGRAM 9.6

```
 10  REM TO ACCESS STUDENT FILE
 20  OPEN  "STUDENT"  FOR INPUT AS FILE #1
 30  FOR K = 1  TO  100
 40  PRINT "Type the ID number of the student you want"
 50  PRINT "Type zero to stop the program"
 70  INPUT I
 80  IF I=0 THEN  170
 90  FOR K = 1 TO 100
100  IF EOF(1) THEN 170
110  INPUT #1,Z,W$,B,H$,D$,G
120  IF Z<>I THEN 150
130  PRINT Z,W$,B,H$,D$,G
140  GO TO 160
150  NEXT K
160  NEXT L
170  CLOSE #1
180  END
```

The output of the above program is as follows:

```
            OUTPUT 9.6

Type the ID number of the student you want
Type zero to stop the program
? 7736
 7736  Bobbi   21   U  MIS   15
Type the ID number of the student you want
Type zero to stop the program
? 4461
 4461  Rod     29   U  ACC   06
Type the ID number of the student you want
Type zero to stop the program
? 0
```

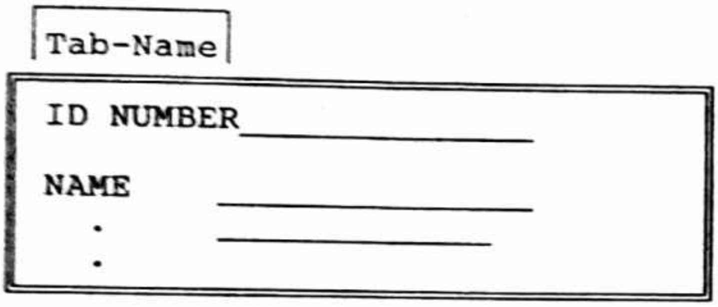
As shown so far, each record has to be read individually one after the other. That is, whenever we used an INPUT or a PRINT statement each record was read one after the other. We could not access the 10th record without reading the first nine records. However, to read the 10th record, this process discussed so far takes a long time. It may not be acceptable in some business or non-business applications. Therefore, an efficient method is needed for accessing record. A random file does help solve this problem.

FILE STORAGE - RANDOM FILE

A random file can access each record directly without accessing the preceding records. A random file works like index cards or a rolodex where a card can be selected without searching every preceding card to see whether it is the

card you want. In each case, the cards are numbered or labeled to hold the specific information. For example,

FIGURE 9.2



The card contains information on a specific supplier.

Each card is a record, and each item of information is a field. In this record, there are many fields. Each record is accessed by locating the tab-name of the card among the cards. In BASIC, the accessing is done by identifying each record using a record number. The first record is considered record number 1 and each subsequent record number is incremented by one. That is, the second record is 2, and so on. This type of random file is called a relative file. The main advantage of a random file is that it allows a record to be moved from disk to CPU without reading each of its preceding records. Each record is stored within preassigned storage areas. Each storage has its own unique identification number. To access a record, we specify the identification number of the record number, the identification number of storage. For example, to locate number 5, we specify number 5.

To create a random file, we must first open a file. For a random file, one OPEN statement is sufficient to allow the creation, appending, or reading of the file. However, each OPEN statement requires that you must determine the length of a record before creating a file. For example, our

file in Table 9.1 is rewritten to indicate the size of each field.

TABLE 9.4

File Name : STUDENT

FIELD NAMES

RECORD#	ID	FIRST NAME	AGE	MARITAL STATUS	MAJOR	CREDIT HOURS	TOTAL
FIELD SIZES	4	10	2	1	3	2	= 22 characters
1	7736	Bobbi	21	U	MIS	15	
2	9753	Carol	20	M	MGT	18	
3	4283	Connie	22	M	MIS	12	
4	6699	Dianne	21	U	ACC	15	
5	3750	Doug	22	M	MIS	12	
6	4321	Jane	22	M	MGT	15	
7	0123	Jeff	22	M	MIS	12	
8	1111	John	22	M	MIS	12	
9	1006	Julie	21	U	MGT	12	
10	1122	Kim	22	M	MIS	12	
11	2048	Kris	22	U	MIS	18	
12	9110	Patrick	22	U	ACC	12	
13	9462	Ricit	22	U	MIS	15	
14	4461	Rod	29	U	ACC	06	
15	5612	Susan	21	U	MIS	16	

Marital Status Code: U = Unmarried M = Married
Major Code MIS = Management Information System
MGT = Management, ACC = Accounting

In this file, ID is specified to be of 4 characters (spaces), FIRST NAME is 10 characters, AGE is 2 characters, MARITAL STATUS IS 1 character, MAJOR is 3 characters, and

CREDIT HOURS is 2 characters. By specifying these sizes for the respective fields, we cannot put a value larger than the specified size. Adding these sizes shows that the total number of characters in each records could, therefore, be up to 22.

Since we know that each record should have a maximum of 22 characters, now we are ready to create a random file of these records.

As usual, to create a file, we must first open the file. The general form of the OPEN statement is:

```
Ln  OPEN "R",#number,"filename", length
```

In this statement, the statement number (Ln) is followed by the Keyword OPEN and the letter R within quotes and a comma. R means that we are opening a random file. The R is followed by the sign (#) and the channel number and a comma. The comma is followed by the filename within quotes and a comma. The comma is followed by the length of the record. The length is the number of characters in each record. It is advisable to create a record of length greater than the actual record size in the file. For example, to create a random for the above file:

```
10  OPEN  "R", #1, "Student1", 30
```

In this statement, the Student file of size 30 characters is opened as a random file.

Some other versions of the OPEN statement are listed in Table 9.5.

TABLE 9.5

OTHER VERSIONS OF THE **OPEN** STATEMENT FOR RANDOM FILE

SYSTEM	STATEMENT
CDC BASIC	none
BASIC PLUS	none
ANSI BASIC	Ln OPEN #number:NAME "filename", ACCESS INPUT or OUTPUT , ORGANIZATION RELATIVE
Microsoft BASIC	Ln OPEN "R",#number, "filename", length
Digital BASIC	Ln OPEN "filename" FOR INPUT or OUTPUT AS FILE # number ORGANIZATION RELATIVE

NOTE: When a word is capitalized, you must use that word exactly. The file name could be a string of up to 8 characters.

Since very few mainframe computers' BASICs create random files, my example will be in the context of personal computers (PC). The general format of the OPEN statement for IBM PC is :

```
Ln  OPEN "filename"  AS channel number LEN = size
```

After opening the file, we must specify the size of each field in the record. The general form of the FIELD statement is:

```
Ln FIELD #number, width AS field name, width AS field name
```

In this statement, the statement number (Ln) is followed by the keyword FIELD and the file number and a comma. The comma is followed by the width of a field which is followed by the keyword AS and the field name. Each field name must be a string variable. All field names of the record must be listed in this statement. The FIELD statement must come after the OPEN statement. For example, for our file, the FIELD statement would be:

```
10 FIELD #1,4 AS I$,10 AS N$,2 AS A$,1 AS S$,3 AS M$,2 AS C$
```

In this statement, the FIELD consists of 4 characters for I$ (ID), 10 characters for N$ (FIRST NAME), 2 characters for A$ (AGE), 1 character for S$ (MARITAL STATUS), 3 characters for M$ (MAJOR), and 2 characters for C$ (CREDIT HOURS). These field sizes were defined in Table 9.4. Notice that all variables in the field statement are string variables. The ID, AGE, and CREDIT HOURS even though they have numeric values are labeled as string variables. That is, all variables in the FIELD statement must be string variables. However, if a value has to be numeric, it should be read as a numeric but converted to a string value for storage and converted back to a numeric after being read from the file as a string. However, this requires a different procedure. Also, the total width of all fields must be less then or equal to the length of the record given in the OPEN statement.

After the FIELD is defined, the data can only then be assigned to each field in the file with either of the following statements.

```
Ln  LSET string var= string expression or value

                        or

Ln  RSET string var= string expression or value
```

In each statement, the statement number (Ln) is followed by the keyword LSET or RSET and a string variable, the same name as defined in the FIELD statement. The field name is followed by the equal sign (=) and the string expression or value. The LSET is used to assign the string expression or the value to left part of the computer memory assigned to the field, called **left-justified**. For example,

```
10  LSET L$= "BOBBI"
```

In this statement, the name BOBBI is read and stored in the first five memory spaces of the ten memory spaces assigned to the field L$. The last five spaces in the field are filled with blank. After this statement, the computer memory assigned to the field L$ would look as follows:

```
   L$
1234567890
----------
BOBBI
----------
```

On the other hand, the RSET assigns the value to the right five spaces of the memory (right justified) and leaves the five first spaces blank. For example,

```
10 RSET  L$="BOBBI"
```

In this statement, the name BOBBI is read and stored in the last five spaces of the ten spaces assigned to the field L$. The first five spaces are filled with blanks. After this statement, the computer memory assigned to the field L$ would look as follows:

```
   L$
1234567890
----------
     BOBBI
----------
```

However, RSET should not be used unless there is a specific reason to move the values to the right of the memory. That is, the LSET should be used in most cases. Also, always use these statements (LSET or RSET) for assigning values to the fields defined in the FIELD statement. Do not use INPUT, LET, or READ statements for this purpose, for they will cause unforeseen problems. However, you do need the INPUT, LET, or READ statement to read data from a file or the keyboard.

As indicated, the LSET and RSET can only use string variables. If numeric data is to be read into a field, it must be converted into a string character before assigning it to a variable in a field defined in the FIELD statement. However, the size of the field defined in the FIELD statement to store numeric data as a string data must be at least four spaces. The following function statement is used for this purpose:

```
Ln LSET string var = MKS$(numeric var or constant)
```

In this statement, the statement number is followed by a keyword LSET and a string variable. The string variable is followed by an equal sign which is followed by the function MKS$ and a numeric value or a numeric variable. The function MKS$ converts the numeric to a four-character string. Therefore, the field size for the variable defined in the FIELD statement must be at least four characters. For example,

```
10 OPEN "R", #1, "TEST", 10
20 FIELD #1, A$ AS 4
30 A = 25
40 LSET A$=MKS$(A)
```

In statement 30, the variable A is assigned number 25. In statement 40, the value of A is stored into the string variable A$ by converting the value to a four-character string. Notice that the field size of A$ is four.

The FIELD statement is also used to identify the fields in the record for retrieval. After a record is retrieved, then the fields defined in the FIELD statement could be assigned to other variables with the LET statement or printed as usual.

However, since all field values are stored as string values, a field containing a numeric value needed to be used as a numeric value must be converted back to numeric value before using it as a numeric value in the program. For this conversion, the function CVS is used. For example, to convert the A$ (AGE) back to numeric:

```
10    LET K = CVS(A$)
```

In this statement, the variable K will now contain the numeric value which was stored as a string value in the string variable A$.

The FIELD statement does not store the records in the file. Therefore, after assigning a value to each field, the record must be stored in the random file. For storing each record in the file, the general format of the statement is :

```
Ln PUT #number, record
```

In this statement, the statement number (Ln) is followed by the keyword PUT and the file number and a comma. The comma is followed by a record number of the record in the file assigned to the record. For example,

```
10 PUT #1, 6
```

In this statement, the record is written as record number 6 in file number 1. Obviously, if the record had been the first record in the file, the record number should have been 1 instead of 6. Therefore, most often instead of using a constant like 1 or 6, a variable is used to change the record numbers. For example,

```
50 K = K+1
60 PUT #1, K
```

In these statements, the variable K works as a counter which keeps a count of the records read. Each time a new record is created the counter changes to account for the new record number.

CREATING A RANDOM FILE

Let us create a program which will create a random file for the file in Table 9.4.

PROGRAM 9.7

```
 10 REM  TO CREATE A RANDOM FILE
 20 OPEN "STUDENTR" AS 1 LEN = 30
 30 FIELD #1,4 AS I$,10 AS S$,4 AS W$,1 AS D$,3 AS M$,4 AS T$
 40 FOR K =1 TO 100
 50 PRINT "Type six values - id, name, age, marital status"
 60 PRINT "major, credit hours per semester. Type zero to stop"
 70 INPUT I,N$,A,S$,M$,C
 80 IF I=0 THEN 170
 90 LSET I$=MKS(I)
100 LSET S$=N$
110 LSET W$=MKS$(A)
120 LSET D$=S$
130 LSET R$=M$
140 LSET T$=MKS$(C)
150 PUT #1,K
160 NEXT K
170 CLOSE #1
180 END
```

In this program, the STUDENTR file is opened in statement 20 with record size of 30 spaces. The FIELD is declared in statement 30. Variable I$ is of size 4, S$ of size 10, A$ of size 4, D$ of size 1, M$ of size 3, and T$ of size 4. Values for each record are inputted in line 70. In lines 90, 110, and 140, the numeric variables I, A, and C are converted into string variables by using MKS$ function. The inputted string variables N$, S$, and M$ are just assigned to the declared fields in the FIELD statement. After assigning each inputted value to the variables in the FIELD statement, the record is written in the file in line 150. Notice that the variable names used in line 70 are different than the variables used in the FIELD statement. In line 150, the PUT statement along with the K variable assigns a record number to each record as they are written in the file. Line 170 closes the file.

The output of the program would look as follows:

OUTPUT 9.7

```
Type six value - id, name, age, marital status
 major, credit hours per semester. Type zeros to stop.
? 7736,Bobbi,21,U,MIS,15
Type six value - id, name, age, marital status
 major, credit hours per semester. Type zeros to stop.
? 9753,Carol,20,M,MGT,18
Type six value - id, name, age, marital status
 major, credit hours per semester. Type zeros to stop.
? 4283,Connie,22,M,MIS,12
Type six value - id, name, age, marital status
 major, credit hours per semester. Type zeros to stop.
? 6699,Dianne,21,U,ACC,15
Type six value - id, name, age, marital status
 major, credit hours per semester. Type zeros to stop.
? 3750,Doug,22,M,MIS,12
```

(continued)

```
                    OUTPUT 9.7  (continued)

Type six value - id, name, age, marital status
 major, credit hours per semester. Type zeros to stop.
? 4321,Jane,22,M,MGT,15
Type six value - id, name, age, marital status
 major, credit hours per semester. Type zeros to stop.
? 0123,Jeff,22,M,MIS,12
Type six value - id, name, age, marital status
 major, credit hours per semester. Type zeros to stop.
? 1111,John,22,M,MIS,12
Type six value - id, name, age, marital status
 major, credit hours per semester. Type zeros to stop.
? 1006,Julie,21,U,MGT,12
Type six value - id, name, age, marital status
 major, credit hours per semester. Type zeros to stop.
? 1122,Kim,22,M,MIS,12
Type six value - id, name, age, marital status
 major, credit hours per semester. Type zeros to stop.
? 0,0,0,0,0,0
```

READING A RANDOM FILE

Once a file is created, it could then be read. The following program shows how to read a random file.

```
                    PROGRAM 9.8

10 REM TO READ THE FILE
20 OPEN "STUDENTR" AS 1 LEN = 30
30 FIELD #1,4 AS I$,10 AS S$,4 AS W$,1 AS D$,3 AS M$,4 AS T$
40 FOR K =1 TO 100
50 GET #1, K
60 LET I=CVS(I$)
70 LET C=CVS(T$)
80 LET A=CVS(W$)
90 PRINT I,N$,A,S$,M$,C
100 NEXT K
110 CLOSE #1
120 END
```

As shown in the above program, the file must be opened as a random file. The FIELD statement defines the field names and sizes. The field sizes must match the sizes defined in the FIELD statement when the file was created. However, to read a random file, use the GET statement. The general format of the GET statement is:

```
Ln    GET #channel number, record number
```

In this statement, the line number (LN) is followed by the keyword GET and a channel number used in the OPEN statement. The channel number is followed by a comma and the record number of the record you want to read. Statement 50 in the above program is an example of the GET statement. This statement will read one record at a time depending upon the value of K.

In the above program, the file is opened to be read. Line 50 reads each record and lines 60 through 80 converts the string values back to numeric values by using CVS function. Line 90 prints each record. The output of the above program is as follows:

OUTPUT 9.8

```
7736,Bobbi,21,U,MIS,15
9753,Carol,20,M,MGT,18
4283,Connie,22,M,MIS,12
6699,Dianne,21,U,ACC,15
3750,Doug,22,M,MIS,12
4321,Jane,22,M,MGT,15
0123,Jeff,22,M,MIS,12
1111,John,22,M,MIS,12
1006,Julie,21,U,MGT,12
1122,Kim,22,M,MIS,12
2048,Kris,22,U,MIS,18
```

In this output, each record was read and printed.

ACCESSING A RECORD IN RANDOM FILE

Most often a certain record is needed to be read and printed instead of the whole file as shown in the previous program. Therefore, let us write a program showing how to access a record in a random file. For example,

PROGRAM 9.9

```
 10 REM  TO SEARCH A RANDOM FILE
 20 OPEN "STUDENTR" AS 1 LEN = 30
 30 FIELD #1,4 AS I$,10 AS S$,4 AS W$,1 AS D$,3 AS M$,4 AS T$
 40 FOR K =1 TO 100
 50 PRINT "Type the record number you want to see"
 60 PRINT " Type zero to stop."
 70 INPUT I
 80 IF I = 0  THEN 170
 90 IF I< 1 THEN 50
100 IF I> 10 THEN 50
110 GET #1, I
120 LET I=CVS(I$)
130 LET C=CVS(T$)
140 LET A=CVS(W$)
150 PRINT I,N$,A,S$,M$,C
160 NEXT K
170 CLOSE #1
180 END
```

In this program, you are to enter the record number, and then the computer searches the file to access the record by matching the record number with the record number in the file. When the record is found, the record is printed. Since we have only ten records in the file, therefore, if you type a number greater than 10 or less than one, the computer asks for the number again.

The output of the program would look as follows:

OUTPUT 9.9

```
Type the record number you want to see
Type zero to stop
? 3
 4283 Connie   22   M     MIS     12
Type the record number you want to see
Type zero to stop
? 8
 1111 John     22   M     MIS     12
Type the record number you want to see
Type zero to stop
? 0
```

UPDATING A RANDOM FILE

Updating means changing existing records or adding new records in the file. In a random file, data can be changed in the existing file or new records could be added to the file. Updating of the existing record could be done by simply retrieving a record and changing values and putting it back. When a new record is stored in the same place, the old record is deleted. Adding a record can take place by simply identifying a new record number where the new record will be place without affecting the existing records. For example, the following program shows how to add new records to the file.

PROGRAM 9.10

```
 10 REM  TO UPDATE A RANDOM FILE
 20 OPEN "STUDENTR" AS 1 LEN = 30
 30 FIELD #1,4 AS I$,10 AS S$,4 AS W$,1 AS D$,3 AS M$,4 AS T$
 40 FOR K =1 TO 100
 50 PRINT "TYPE RECORD NUMBER. TYPE ZERO TO STOP"
 60 INPUT L
 70 IF L=0 THEN 210
 80 PRINT "Type six values - id, name, age, marital status"
 90 PRINT "major, credit hours per semester."
100 INPUT I,N$,A,S$,M$,C
110 LSET I$=MKS(I)
120 LSET S$=N$
130 LSET W$=MKS$(A)
140 LSET D$=S$
150 LSET R$=M$
160 LSET T$=MKS$(C)
170 PUT #1,L
180 NEXT K
190 CLOSE #1
200 END
```

In the program, beside inputting data in the record, you are also asked to input the record number. This number is needed for storing the record in a particular location. However, keep in mind you must remember the number of records you had entered previously. If in case you enter the number of the existing record, the computer replace the old record with the new record.

The output of the program is as follows:

OUTPUT 9.10

```
TYPE RECORD NUMBER. TYPE ZERO TO STOP
? 11
Type six values - id, name, age, marital status
major, credit hours per semester.
? 2048,Kris,22,M,MIS,12
TYPE RECORD NUMBER. TYPE ZERO TO STOP
? 12
Type six values - id, name, age, marital status
major, credit hours per semester.
? 9110,Patrick,22,U,ACC,12
TYPE RECORD NUMBER. TYPE ZERO TO STOP
? 13
Type six values - id, name, age, marital status
major, credit hours per semester.
? 9462,Ricit,22,U,MIS12
TYPE RECORD NUMBER. TYPE ZERO TO STOP
? 14
Type six values - id, name, age, marital status
major, credit hours per semester.
? 4461,ROD,29,U,ACC,06
TYPE RECORD NUMBER. TYPE ZERO TO STOP
? 15
Type six values - id, name, age, marital status
major, credit hours per semester.
? 5612,Susan,21,U,MIS,16
TYPE RECORD NUMBER. TYPE ZERO TO STOP
? 0
```

All of the examples of the random file were based on the relative record concepts. That is the first record was number 1, the second record was number 2, and so on. Therefore, this means that you must remember the record number of

record in the file. This requires keeping a list of records on a piece of paper to make reference about number of records or listing file to find out the record locations. However, this method of locating a record is inconvenient for many users. Therefore, a better method for locating a record would be to use the primary key field of each record. For example, the ID number of a student or an employee or a part number of an inventory could be used as a reference to a record. Therefore, it is better to use the primary key as a record number. However, the disadvantage of this method is that a file with ten records may require a file of size 9999 records if a key field is of size four digits. For example, by storing our file under the primary key method, we will use ID field as a key. In this field, each record has a four-digit number. Therefore, we will be creating a file of size 9999 records, even though we have ten records in the file. Thus, a record with an ID number 9753 will be placed in the 9753th position of the file, and a record with ID number 1111 will be stored in the 1111st position of the file.

The following program shows the primary key method for storing a file:

PROGRAM 9.11

```
 10 REM  TO CREATE A RANDOM FILE
 20 OPEN "STUDENTR" AS 1 LEN = 30
 30 FIELD #1,4 AS I$,10 AS S$,4 AS W$,1 AS D$,3 AS M$,4 AS T$
 40 FOR K =1 TO 100
 50 PRINT "Type six values - id, name, age, marital status"
 60 PRINT "major, credit hours per semester. Type zero to stop"
 70 INPUT I,N$,A,S$,M$,C
 80 IF I=0 THEN 180
 90 LSET S$=N$
100 LSET W$=MKS$(A)
110 LSET D$=S$
120 LSET R$=M$
130 LSET T$=MKS$(C)
140 PUT #1,I
150 NEXT K
160 CLOSE #1
170 END
```

In this program, the value of the variable I is used as a record number. Since this record number could always be printed, we do not have to store it as a field in the file. The output of this programs is follows:

OUTPUT 9.12

```
Type six value - id, name, age, marital status
 major, credit hours per semester. Type zeros to stop.
? 7736,Bobbi,21,U,MIS,15
Type six value - id, name, age, marital status
 major, credit hours per semester. Type zeros to stop.
? 9753,Carol,20,M,MGT,18
Type six value - id, name, age, marital status
 major, credit hours per semester. Type zeros to stop.
? 4283,Connie,22,M,MIS,12
Type six value - id, name, age, marital status
 major, credit hours per semester. Type zeros to stop.
? 6699,Dianne,21,U,ACC,15
Type six value - id, name, age, marital status
 major, credit hours per semester. Type zeros to stop.
? 3750,Doug,22,M,MIS,12
Type six value - id, name, age, marital status
 major, credit hours per semester. Type zeros to stop.
? 4321,Jane,22,M,MGT,15
Type six value - id, name, age, marital status
 major, credit hours per semester. Type zeros to stop.
? 0123,Jeff,22,M,MIS,12
Type six value - id, name, age, marital status
 major, credit hours per semester. Type zeros to stop.
? 1111,John,22,M,MIS,12
Type six value - id, name, age, marital status
 major, credit hours per semester. Type zeros to stop.
? 1006,Julie,21,U,MGT,12
Type six value - id, name, age, marital status
 major, credit hours per semester. Type zeros to stop.
? 1122,Kim,22,M,MIS,12
Type six value - id, name, age, marital status
 major, credit hours per semester. Type zeros to stop.
? 0,0,0,0,0,0
```

Let us write a program to read the file created above:

PROGRAM 9.12

```
 10 REM  TO ACCESS A RECORD IN RANDOM FILE
 20 OPEN "STUDENTK" AS 1 LEN = 30
 30 FIELD #1,4 AS I$,10 AS S$,4 AS W$,1 AS D$,3 AS M$,4 AS T$
 40 FOR K =1 TO 100
 50 PRINT "Type the ID number of the student"
 60 PRINT " Type zero to stop."
 70 INPUT I
 80 IF I = 0 THEN 170
 90 IF I< 1 THEN 50
100 IF I> 9999 THEN 50
110 GET #1, K
120 IF  I<>K THEN PRINT "THE RECORD DOES NOT EXIST"
130 IF  I<>K THEN 50
140 LET C=CVS(T$)
150 LET A=CVS(W$)
160 PRINT I,N$,A,S$,M$,C
170 NEXT K
180 CLOSE #1
190 END
```

In this program, the ID number is entered and the file is searched to check whether such a record exists. If it exists, the record is printed and if the record does not exist, the message is printed and you are asked to enter another record number. The output of the above program is as follows:

(next page)

OUTPUT 9.13

```
Type the ID number of the student
Type zero to stop
? 9642
 9642 Ricit          22    U   MIS     15
Type the ID number of the student
Type zero to stop
? 7736
 7736 Bobbi          21    U   MIS      15
Type the ID number of the student
Type zero to stop
? 0
```

QUESTION

1. What is a computer file, and why do we need a computer file?

2. Define a field in a computer file. What is a field used for? Give some examples of a field.

3. Define a record in a computer file. What does a record represent?

4. What is a key field and why is it needed?

5. Can any field be used as a key field?

6. What are the differences between a sequential and an nonsequential file?

7. Discuss situations where one will use sequential files.

8. What are the steps needed to create a nonsequential file?

9. What are the steps needed to create a sequential file?

10. List three types of OPEN statements.

11. What are the differences between the PRINT and the WRITE statements?

12. Are the steps in creating a sequential file different than from the steps in creating a nonsequential file? Discuss with examples.

13. Create a customer file for a company with the following fields.

Field Name	Field Type
Name	Character
Street Address	Character
City	Character
District	Character

P.O. Office Code	Numeric
Province	Character
Type of Business	Character
Credit Rating Code	Numeric (Good = 1, Bad = 0, Delinquen
Size of Business	Numeric (Sales in millions of rupees)
Sales Quantity	Numeric (Quantity the customer buys)
Sales Amount	Numeric (Amount in rupees)

Create ten customers and store the records in the file.

14. Write a program that updates the above file. Show the output by adding three new customers and print the whole file.

15. Write a program that deletes records. Show the output by deleting the records of the first and the fourth customers from the file, who have gone out of business.

16. Write a program that locates records in the file. Show the output by locating the name of the third customer.

17. The customer in the fifth record in the file moved from one province to another; change the address of the customer in the record. Print the whole file to make sure that the change has been made correctly.

18. Create a random customer file for a company with following fields.

Field Name	Field Type	Field Size
Last Name	Character	15
First Name	Character	15
Street Address	Character	25
City	Character	15
District	Character	10
P.O. Office Code	Numeric	5
Province	Character	5

Type of Business	Character	10
Credit Rating Code (Good = 1, Bad = 0, Delinquent = 0)	Numeric	1
Size of Business (Sales in millions of rupees)	Numeric	10
Sales Quantity (Quantity the customer buys)	Numeric	6
Sales Amount (Amount in rupees)	Numeric	10

Create ten customers and store the records in the file.

19. Write a program that updates the above file. Show the output by adding three new customers and print the whole file.

20. Write a program that deletes records. Show the output by deleting the records of the first and the fourth customers from the file, because they have gone of out of business.

21. Write a program that locates records in the file. Show the output by locating the name of the third customer.

22. The fifth customer in the file moved from one province to another. Change the address of the fourth customer. Print the whole file to make sure that the change has been made correctly.

23. What is a random file? Is a random file better than a sequential file?

24. Are the steps required to create a random file different from the steps to create a nonsequential file?

25. What are the differences in creating a sequential file and creating a random file?

26. Write a program that copies an existing sequential file to another sequential file.

27. Write a program that stores data in a file for the following fields and processes payroll at end of the month. The program should accept hours worked during the month and then should calculate gross pay and income tax deductions for each employee.

Field Name
Last Name
First Name
Department Number, where the employee works
Street Address
City
P.O. Code
Rate Per Hour
Employee Code

28. Write a program that keeps an account of your household budget. The program should use a file that has three fields: the budget item name, the monthly budget for the item, and the monthly expenses for each item. For example, one of the budget items named could be rent for which you have budgeted Rs. 2000.00 a month. The program should ask for actual monthly expenses and should update the file and should print a monthly report showing your budget item name, monthly budgeted expenses, and actual monthly expenses.

INDEX

A

B

C

D

E

H

I

K

L

M

P

Q

R

About the Book

The contemporary world is in the midst of a Computer Revolution that has overshadowed the Industrial Revolution. The computer revolution has already affected everyday life and has greater potential to impact lives of people in many predictable and unpredictable ways. Its effect on human life and economy are now more manifestly visible in industry, education, health, agriculture, communication and a host of other significant areas.

This wonderful book provides an introduction to the BASIC programming language, starting out with a general discussion of the computer hardware and software.

Systematically divided into nine selfcontained chapters the book provides an easy-to-understand and easy-to-apply material for all those interested in the mysteries and marvels of the almighty computer.

About the Author

The author, Dr. Syed Shahabuddin, holds the highest professional qualifications in the field and is presently serving on the staff of a reputed American university as Professor of Management Information System and Management Science.

FEROZSONS (Pvt.) LTD.
LAHORE-RAWALPINDI-KARACHI

Rs. 80.00

969 0 10004 1